MS-DOS® Series

For Personal Computers Running the MS-DOS®

Version 1.0

Microsoft® Visual Basic™

Programming System for MS-DOS®

マイクロソフト ビジュアル ベーシック　プログラミング システム

プログラミング ガイド

Microsoft® VISUAL BASIC™

Microsoft® Visual Basic™

Programming System for MS-DOS®

マイクロソフト ビジュアル ベーシック　プログラミング システム

プログラミング ガイド

Microsoft® VISUAL BASIC™

ご注意

① このソフトウェアの著作権は、米国マイクロソフト社にあります。
② このソフトウェアおよびマニュアルの一部または全部を無断で使用、複製することはできません。
③ ソフトウェアは、コンピュータ 1 台につき 1 セット購入が原則になっております。
④ このソフトウェアおよびマニュアルは、本製品の使用許諾契約書のもとでのみ使用することができます。
⑤ このソフトウェアおよびマニュアルを運用した結果の影響については、いっさい責任を負いかねますのでご了承ください。

MS, Microsoft, MS-DOS は米国マイクロソフト社の登録商標です。
QuickBasic, Visual Basic, Windows は米国マイクロソフト社の商標です。
IBM は米国 IBM 社の登録商標です。
Intel は米国インテル社の登録商標です。
Lotus は米国ロータス・ディベロップメント社の登録商標です。

Microsoft Visual Basic for MS-DOS のマニュアルは、Microsoft Word for Windows Version 1.2A を使用して製作しました。

目次

はじめに xv

第 1 部　Visual Basic の使い方

1　Visual Basic のインストールと実行 3
1.1　Visual Basic の起動 4
1.2　Visual Basic の概要 5
1.3　Visual Basic のウィンドウ 7
ウィンドウの操作方法 8
1.4　フォーム デザイナの起動 10
フォーム デザイナの画面の構成 12
プログラミング環境へ戻る 12
1.5　Visual Basic Help の利用 12
ヘルプの表示方法 13
ハイパーリンクを使った関連するトピックの表示 13
その他のヘルプの表示方法 14
マウスの右ボタンを使ったヘルプの表示 15
ヘルプのプログラム例のカットとペースト 15
1.6　Visual Basic の利用法を学ぶ 16

2　Hello, world アプリケーションの作成 17
2.1　ステップ 1: プロジェクトとフォームの作成 18
2.2　ステップ 2: フォームへのコントロールの配置 19
2.3　ステップ 3: プロパティの設定 21
2.4　ステップ 4: コードの入力 23
2.5　ステップ 5: アプリケーションの保管と実行 25
2.6　実行ファイルの作成方法 25
2.7　アプリケーションの修正 26

3　イベントドリブン アプリケーションの作成 29
3.1　イベントドリブン アプリケーションの動作 31
3.2　イベントドリブンと従来の Basic との比較 32
従来の Basic アプリケーション 34
イベントドリブン アプリケーション 34

3.3 インターフェイスの設計 35
Visual Basic のオブジェクト 35
フォームの作成と編集 36
フォームへのコントロールの配置 37
オブジェクトの選択 38
オブジェクトの移動とサイズ変更 39
コントロールとフォームの削除 40
3.4 プロパティの設定 40
フォーム デザイナにおけるプロパティの設定 41
プログラム中からのプロパティの設定 43
プロパティの設定方法の比較 43
サイズと位置を決めるプロパティの設定 45
3.5 Basic コードの記述 45
プロシージャを使った操作 45
イベント プロシージャの作成 47
モジュールレベル コードに対する宣言と定義の追加 49
Form_Load プロシージャへのコードの追加 50
プログラムの終了方法 51
メソッドの利用 51
キーボード インターフェイスのサポート 52
3.6 オブジェクトのフォーカスの取得と設定 55

4 コントロールの作成と利用 59
4.1 コントロールの名前の付け方 60
4.2 コントロールの親子関係 61
4.3 コマンド ボタンに対する応答 62
コマンド ボタンの例 63
4.4 テキスト ボックスを使った入力の取得と出力の表示 64
投資計算のプログラム例 65
4.5 ラベルを使ったテキストの表示 67
温度単位変換のプログラム例 67
4.6 オプション ボタンとチェック ボックスを使った選択項目の設定 68
オプション ボタンの利用 69
チェック ボックスの利用 72
4.7 リスト ボックスとコンボ ボックスによる値の設定 75
リストへの項目の登録 78
指定位置への項目の追加 78
リストからの項目の削除 79
リスト項目の操作 79

リスト ボックスのイベント 83
コンボ ボックスのイベント 86
4.8 フレームによるコントロールのグループ化 87
4.9 ピクチャー ボックスの利用 88
4.10 スクロール バーによる入力の制御 90
スクロール バーの Value プロパティの利用 92
4.11 実行時におけるコントロールの操作 92
実行時におけるコントロールの移動 92
実行時におけるコントロールのサイズ変更 96
実行時におけるコントロールの表示/非表示の切り替え 97
4.12 コントロール配列の利用 98
設計時におけるコントロール配列の作成 99
実行時におけるコントロール配列の作成 100
4.13 タイマー コントロールの利用 102
フォームへのタイマー コントロールの配置 103
タイマー コントロールの初期化 103
タイマー イベントへの応答 104
アラーム クロックの作成 105

5 ダイアログ ボックスの作成 107
5.1 モーダル ダイアログ ボックスとモードレス ダイアログ ボックス 108
5.2 定義済みのダイアログ ボックスの利用 109
INPUTBOX$ による入力の取得 109
MSGBOX によるメッセージの表示 110
コモン ダイアログ ツールキットの利用 112
5.3 独自のダイアログ ボックスの作成 113
フォームの作成とカスタマイズ 114
コントロールの追加 115
複数のフォームの表示と消去 118
$FORM メタコマンド 119

6 プロジェクトの管理 121
6.1 プロジェクトの作成，オープン，保管 122
実行可能ファイルの作成 124
プロジェクトのクローズ 126
6.2 Visual Basic のファイルとモジュール 126
コード モジュール 127
フォーム モジュール 128
バイナリ形式とテキスト形式 128
インクルード ファイル 129

ライブラリ ファイル 130
テキスト ファイル 131
6.3 実行開始ファイルの設定 133
6.4 フォームとコードの印刷 133

7 テキストの処理 135
7.1 テキスト ボックスの使い方 135
複数行のテキスト ボックスとワードラップ 135
テキスト ボックスに入力されたデータの処理 137
日本語の扱い 140
日本語入力システムの制御 140
文字列に含まれるダブル クォーテーションの扱い 141
7.2 CLIPBOARD オブジェクトの使い方 141
テキストの複写と貼り付け 142
テキストの選択 142
テキスト ボックス用のカット & ペースト機能の例 143
クリップボードのクリア 146
7.3 フォームおよびピクチャー ボックスへの出力 146
データの区切りと出力位置 147
桁位置の設定 149
出力位置の指定 149
テキストの高さと幅 150
7.4 数値，日付，時刻の書式の指定 151
数値データの書式の指定 151
日付と時刻の書式の指定 152
7.5 プリンタへの出力 152
出力先の指定 152
プリンタへのテキストの出力 153
フォームの出力 153

8 メニューの作成 155
8.1 メニュー インターフェイスの作成 156
メニュー コントロールのプロパティの設定 158
8.2 メニュー コントロールに対応するコードの記述 158
8.3 Try Menus アプリケーションの作成 159
8.4 セパレータ バーの使い方 162
8.5 アクセス キーとショートカット キーの設定 163
8.6 メニュー コントロールを無効にする 164
8.7 チェック マークの表示 165
8.8 メニュー コントロールを非表示にする 165

8.9 メニューの階層化 165
8.10 アクセス キー，メニュー コントロール，タイトル バーの色設定..... 166
8.11 トップ レベルのメニュー コントロールに対する
Click イベント プロシージャ 167
8.12 実行時のメニュー コントロールの追加と削除..... 167
8.13 マルチ ドキュメント インターフェイス(MDI)..... 173
MDI アプリケーションの作成 176
MDI デスクトップ アプリケーションの作成 179
メモ帳アプリケーション 182

9 エラー ハンドリング 183
9.1 エラー処理が必要な理由 183
9.2 エラー処理の方法 185
エラー処理ルーチンの設定 185
エラー処理ルーチンの記述 186
RESUME を使ったエラー処理ルーチンの終了..... 186
呼び出しパス 189
9.3 モジュール レベルのエラー処理..... 190
プロシージャ レベルとモジュール レベルのエラー処理の使い分け 191
呼び出しパスからのエラー処理ルーチンの検索方法 192
エラー処理ルーチンからの復帰 193
複雑な構成のプログラムのためのガイドライン 194
9.4 エラーの集中処理 195
9.5 エラー処理を後で実行 197
エラー処理を後で実行するためのガイドライン 199
9.6 エラー処理を明示的に無効にする 200
プロシージャ レベルのエラー処理を無効にする..... 200
モジュール レベルのエラー処理を無効にする..... 200
9.7 エラー処理の別の利用法 201
エラー処理をプログラムの制御に利用する 201
エラー処理を情報の取得に利用する 201
ユーザー定義のエラー コード..... 202
9.8 エラー処理を含むコードのコンパイル 202

10 ファイルシステム コントロールの利用 203
10.1 ファイルシステムのためのコントロール 203
ドライブ リスト ボックス 204
ディレクトリ リスト ボックス 205
ファイル リスト ボックス 206
複数のコントロールの同期 207

10.2 ファイル検索アプリケーション 208
SEEK のコード 209
10.3 コモン ダイアログ ツールキット 214
[オープン] ダイアログ ボックス 216
ファイル リスト ボックスの FileName プロパティ 220
ファイル リスト ボックスの Pattern プロパティ 221

第 2 部　プログラミング テクニック

11　デスクトップ出力とキーボード入力 225
11.1 デバイスのサポート 225
11.2 デスクトップ 226
11.3 デスクトップへのテキスト表示 227
デスクトップ上のテキストの表示位置の指定 227
PRINT ステートメントによるテキストと数値の表示 228
PRINT USING による出力の書式指定 230
スペースによるスキップと任意の桁への移動 230
テキスト ビューポートの設定 231
11.4 フォームを使ったアプリケーションでの
デスクトップへのテキスト表示 233
11.5 キーボードからの入力 233
INPUT ステートメント 234
LINE INPUT ステートメント 235
INPUT$ 関数，INPUT¥ 関数 236
INKEY$ 関数 236
テキスト カーソルの制御 238
カーソル位置情報の取得 239

12　ファイルとデバイスの操作 241
12.1 データ ファイルの操作 241
データ ファイルの構成 241
シーケンシャル アクセス ファイルとランダム アクセス ファイル 242
ファイル アクセスの種類 242
ファイル アクセスの概要 243
データ ファイルのオープン 244
データ ファイルのクローズ 249
12.2 シーケンシャル アクセス ファイルの利用 249
簡易テキスト エディタ 250
テキスト エディタの動作 252
テキスト ボックスでの [TAB] キーの効果 257

12.3 ランダム アクセス ファイルの利用 258
ランダム アクセス ファイルのレコード 258
ランダム アクセス ファイルのデータの作成と追加 258
12.4 バイナリ ファイルの I/O 261
バイナリ アクセスとランダム アクセスの違い 261
SEEK によるファイル ポインタの移動 263
シーケンシャル アクセスとランダム アクセスでの SEEK の使い方 266
12.5 デバイスの操作 266
デバイス I/O とファイル I/O の違い 267
シリアル ポートを利用した通信 268
端末エミュレータ アプリケーション 269

13 マウス，キーボード，ドラッグ & ドロップ イベント 271
13.1 マウス イベント 271
マウスに関するイベントの発生と認識 272
マウスの移動と座標 273
マウス ポインタ 274
13.2 マウス イベントの使い方 275
マウスの動きの追跡 275
MouseMove の制限 276
13.3 *Button* 引数と *Shift* 引数 276
マウスのボタンの状態の検出 277
[SHIFT]，[CTRL]，[メニュー] キーの状態の検出 279
13.4 マウス イベントの例 280
13.5 キーボード イベント 284
KeyPress イベント 285
KeyDown イベントと KeyUp イベント 285
KeyCode 引数 286
Shift 引数 287
フォーム全体に有効な処理ルーチン 287
13.6 ドラッグ & ドロップ 288
自動ドラッグ モード 289
オブジェクトのドロップ 289
ドラッグ時のオブジェクトの強調表示 290
ドラッグの開始と終了の制御 291
コントロールの移動 292
コントロール自身へのドロップ 293
13.7 フォームが表示されていないときのマウスの利用 294

14 アプリケーションの外観の変更 297
14.1 ControlPanel プロパティの使い方 297
ControlPanel 配列への値の代入 298
14.2 ControlPanel サンプル アプリケーション 301

15 プログラムの流れの制御 303
15.1 ステートメントの実行順序の変更 303
15.2 論理式 303
15.3 条件判断 307
1 行形式の IF...THEN...ELSE 307
ブロック形式の IF...THEN...ELSE 308
SELECT CASE ステートメント 309
15.4 ループ構造 313
FOR...NEXT ループ 313
DO...LOOP ループ 317
WHILE...WEND ループ 319

16 プロシージャによる構造化プログラミング 321
16.1 プロシージャ 321
イベント プロシージャ 321
ジェネラル プロシージャ 322
複数のモジュールからなるプログラムでのプロシージャの使い方 323
プロシージャへのデータの引き渡し 324
ローカル変数とグローバル変数 324
プロシージャの定義 324
FUNCTION プロシージャの呼び出し 326
SUB プロシージャの呼び出し 326
16.2 プロシージャへの引数の引き渡し 327
パラメータと引数 327
定数と式の引き渡し 329
単純変数の引き渡し 329
配列変数の引き渡し 329
ユーザー定義型変数の引き渡し 330
関数呼び出しを含む式の引き渡し 332
未知の型の引数の引き渡し 332
フォームとコントロールの引き渡し 332
16.3 参照による引き渡しと値による引き渡し 332
引数の参照による引き渡し 332
引数の値による引き渡し 333

配列，ユーザー定義型の引き渡しの方法 335
引数に式として指定できない項目 335
16.4 DECLARE による引数のチェック 335
DECLARE ステートメントが自動生成されない場合 337
TYPE ステートメントと DECLARE ステートメントの位置関係 338
インクルード ファイルによる宣言 339
Quick ライブラリのプロシージャの宣言 342
16.5 変数の共有 342
モジュール内の特定のプロシージャ間での変数の共有 343
モジュール内の全プロシージャでの変数の共有 344
変数宣言の優先順位 346
モジュール間での変数の共有 347
他のモジュールとのプロパティの共有 349
変数の重複 349
16.6 自動変数と STATIC 変数 350
ローカル変数の値の保存 350
16.7 再帰プロシージャ 351
階乗の計算 351
スタックサイズの調整 353

17 アプリケーションのデバッグ 357
17.1 アプリケーションのエラー 357
構文エラー 358
ランタイム エラー 358
論理エラー 359
17.2 デバッグ機能 360
17.3 ヘルプの使い方 364
エラー時のヘルプの表示 364
変数の値とプロシージャの値の表示 364
定数の値の表示 365
17.4 アプリケーションの設計 365
設計のテクニックとデバッグ テクニック 366
17.5 変数の明示的な宣言 367
暗黙の変数宣言 367
変数のデフォルトのデータ型 367
OPTION EXPLICIT ステートメントを使ったデバッグ 368
17.6 問題のあるステートメントでの実行の停止 368
ランタイム エラーの修正と継続 369
ブレークポイントによる実行の中断 370

ウォッチ ポイントによる実行の中断 371
STOP ステートメントによるアプリケーションの中断 372
実行を中断したあとの操作 372
17.7 ウォッチ式の追加 373
プロパティの値の監視 374
17.8 アプリケーション中の指定した部分の実行 374
ステートメントのシングル ステップ実行 375
プロシージャ ステップ実行 375
次に実行するステートメントの設定 376
ヒストリのトレース 377
17.9 イミディエイト ウィンドウの使い方 377
イミディエイト ウィンドウのスコープ 378
イミディエイト ウィンドウに入力できるステートメント 378
17.10 イミディエイト ウィンドウの特性 379
変数やプロパティへの値の代入 379
イミディエイト ウィンドウからのプロシージャの呼び出し 379
イミディエイト ウィンドウからの表示 380
イミディエイト ウィンドウで使えるキーとその用途 380
17.11 コマンドライン引数のテスト 381
17.12 イベントのデバッグに関する注意点 382
トップ レベルのメニュー クリックに対する
プロシージャのデバッグ 382
MouseDown イベント中の実行の中断 382
KeyDown イベント中の実行の中断 382
イベント ドリブン プログラムに特有のバグ 382
17.13 デバッグにおけるその他の問題点 384
同じ名前を持つ変数のスコープ 384
AS ANYとして宣言されたパラメータ 385

18 Quick ライブラリの作成 387
18.1 Quick ライブラリとは 387
Quick ライブラリとオブジェクト モジュール ライブラリ 388
デフォルトの Quick ライブラリ (VBDOS.QLB) 388
コモン ダイアログの Quick ライブラリ (CMNDLG.QLB) 388
18.2 Quick ライブラリのロードと表示 389
Quick ライブラリのロード 389
Quick ライブラリの内容の表示 389
18.3 Quick ライブラリの作成 390
Quick ライブラリの作成に必要なファイル 391
プログラミング環境からの Quick ライブラリの作成 392

コマンドラインからの Quick ライブラリの作成 394
Basic 以外の言語モジュールからの Quick ライブラリの作成 395
18.4 Quick ライブラリのプログラミング上の注意 396
B_OnExit ルーチン 396
文字列と配列の扱い 397
コンパクトな実行可能ファイルの作成 397

付録

A ASCII フォームのフォーマット 401
A.1 記録される情報 401
バージョン番号 402
フォーム情報 402
コントロール ブロック 403
プロパティ 404
Basic コード 406
A.2 ASCII フォームのサンプル 406
A.3 ASCII フォームのロードと保管 409
ファイルのロード時のエラー 409

B メモリ管理 411
B.1 利用可能メモリの設定 411
プログラミング環境のサイズ 411
XMS メモリと EMS メモリの使い方 412
B.2 Visual Basic for MS-DOS のメモリの使い方 414
プログラミング環境のメモリ マップ 415
実行可能 (.EXE) ファイルのメモリ マップ 417
B.3 データとオブジェクトのサイズ 419
データ型 419
ユーザー定義型のオーバーヘッド 420
非実行可能ステートメントのオーバーヘッド 420
配列と文字列のオーバーヘッド 420
フォーム，コントロール，モジュールのオーバーヘッド 421

C プログラム作成上のヒント 423
C.1 サイズを小さくするためのヒント 423
C.2 高速化するためのヒント 424
高速化のためのデータの扱い 424
高速化のための制御構造の最適化 425

ループの高速化 427
入出力の速度の改善 427
高速化のためのその他のヒント 428
C.3 その他のヒント 428

D Visual Basic のメニュー 431
D.1 [ﾌｧｲﾙ] メニュー 431
D.2 [編集] メニュー 432
D.3 [表示] メニュー 433
D.4 [検索] メニュー 433
D.5 [実行] メニュー 434
D.6 [ﾃﾞﾊﾞｯｸﾞ] メニュー 435
D.7 [ﾂｰﾙ] メニュー 436
D.8 [補助] メニュー 436
D.9 [ｳｨﾝﾄﾞｳ] メニュー 437
D.10 [ﾍﾙﾌﾟ] メニュー 438

索引 439

はじめに

Visual Basic Programming System for MS-DOS (以下Visual Basic for MS-DOS) は、MS-DOS 上でイベントドリブン プログラミング モデルとグラフィカル ユーザー インターフェイス (GUI) を実現する、新しいタイプのプログラミング環境です。この環境では、マウスを使って、画面上にボタンやテキスト ボックスなどのオブジェクトを配置し、それぞれに対する動作を Basic 言語で記述するだけで、Microsoft Windows に近い操作性を持った高度なプログラムを作成できます。

高度な生産性

Visual Basic for MS-DOS は、フォーム デザイナと呼ばれる画面設計ツール (FD.EXE) と、コードの記述やデバッグを行うプログラミング環境 (VBDOS.EXE) の 2 つのプログラムによって構成されています。この 2 つのプログラムにより、洗練された MS-DOS アプリケーションを短時間で作成できます。

アプリケーションのユーザー インターフェイスとなる画面の設計は、Visual Basic for MS-DOS のフォーム デザイナで行います。画面上にフォームと呼ばれるウィンドウを作り、ボタン，テキスト ボックス，リスト ボックス，スクロール バーなどのオブジェクトを配置します。それぞれのオブジェクトの基本的な動作は、あらかじめ Visual Basic で定義されているため、マウスでの操作やテキストの処理といった細かい部分のプログラミングは不要です。プログラマは、ボタンが押されたらどうするか、入力されたテキストをどう処理するかなどを、イベント (例えばボタンをクリックした、キーが押されたなど) に対応した形でプログラムすればよいのです。

プログラミング環境は高度な編集機能とデバッグ機能を備えており、フォーム デザイナで作成した画面上のオブジェクトに関連づけながらプログラムを編集することができます。プログラミング環境では、デバッグを簡単にするために、インタープリタがプログラムを 1 行ずつ実行しています。

完成したプログラムは、「EXE ファイル作成」機能によって実行形式のファイルに変換 (コンパイル) できます。これにより、Visual Basic for MS-DOS がなくても実行できる、独立したアプリケーションが作成できます。

フォーム デザイナ (FD.EXE) とプログラミング環境 (VBDOS.EXE) は別々のプログラムですが、お互いを自動的に呼び出す機能を持っているので、コードの記述とフォームの設計を簡単に切り換えることができます。また、カーソルの位置の単語を自動的に検索して関連する情報を表示するオンライン ヘルプが用意されており、MS-DOS アプリケーションの開発を支援します。

Visual Basic for MS-DOS では、ユーザーに入力を求めたり、メッセージを表示したりといった定形的な処理を行うためのダイアログ ボックスがあらかじめ用意されています。コモン ダイアログ ボックスのツールキット (CMNDLG.BAS, CMNDLGF.FRM) を使うと、ファイルのオープン，保管，印刷など、よく使われる 7 種類のダイアログ ボックスを簡単に表示できます。また、これとは別に、**MSGBOX** と **INPUTBOX$** 関数を使

うと、メッセージを表示したり、ユーザーに入力を求めるときに使う定形のダイアログ ボックスを表示できます。これらの関数を利用すれば、コードを 1 行付け加えるだけで、ユーザーとの対話の機能を追加できます。

Visual Basic for MS-DOS の強力な機能

Visual Basic for MS-DOS では、さまざまな種類の MS-DOS アプリケーションを作成するのに必要な機能が提供されています。プログラミング環境は XMS メモリや EMS メモリに対応しており、これらのメモリ領域を使ったオーバーレイを自動的に行うため、作成するアプリケーション用に大きなメモリ領域を残します。また、すべての文字列変数は far メモリに格納されるため、変数領域の不足によるエラーの発生が少なくなっています。QuickBasic では変数領域の不足で実行できなかったプログラムも、Visual Basic for MS-DOS では、多くの場合に実行可能です。

言語仕様

Microsoft Basic Professional Development System version 7.1 から導入された多くの強力な言語機能をサポートしているため、より簡単にコードを書くことができます。

- **REDIM PRESERVE** を使うと、データを保持したまま、配列のサイズを変更できます。
- 固定小数点数を表現する **CURRENCY** (通貨) データ型を使うと、丸めに関する誤差がなくなると同時に、演算速度を高めることができます。
- 日付，時刻，表示書式に関する組み込みの関数によって、日付や時間の計算を簡単に行えます。
- プロシージャ レベルでエラー処理を行うことができます。
- ユーザー定義型に静的配列を含めることができます。

さらに、新しく追加された **OPTION EXPLICIT** ステートメントを使うと、Basic の暗黙の宣言が禁止になり、すべての変数は宣言しなければ使用できなくなります。これによって、コード中のタイプ ミスやデータ型の不一致などによるエラーを未然に防ぐことができます。

Visual Basic for MS-DOS では、フォームをテキスト ファイル (ASCII フォーム) に保存できます。このファイルは、エディタで修正し、再び Visual Basic for MS-DOS で読み込むことができます。

マルチ ドキュメント インターフェイス (MDI) 機能を使えば、1 つのフォームの中にさらに複数のフォームを開くこともできます。

最適化されたネイティブ コード コンパイラによって、高速なスタンドアロンの MS-DOS アプリケーションを作成できます。

他の Basic との互換性

Visual Basic for MS-DOS は、QuickBasic で作成されたプログラムを読み込み，実行することができます。Visual Basic for MS-DOS のコンパイラを使って QuickBasic のプログラムをコンパイルすると、コードが最適化され、文字列の処理機能が強化されます。また、MS-DOS 用に設計された Microsoft Basic Professional Development System version 7.1 のプログラムもロード，実行できます。これらのプログラムに、フォームやコントロールを使ったグラフィカルなユーザー インターフェイスを加えることによって、既存のアプリケーションの操作性を一新することができます。このとき、プロシージャなどの既存のコードは、多くの場合に流用可能です。さらに、多くの新しい言語仕様を利用すれば、既存のコードの改良も行えます。

基本的な用語

初めてイベントドリブン プログラミングを行うときは、聞き慣れない用語がいくつかあると思います。ここでは、イベントドリブン プログラミングに関して、Visual Basic for MS-DOS で使う基本的な用語を説明します。

アタッチ

コントロールまたはフォームにコードを対応させる動作です。フォームまたはコントロールにイベント プロシージャをアタッチするというような使い方をします。

イベント

フォームやコントロールが認識可能な動作をイベントと呼び、Visual Basic では、このイベントに対する応答方法を記述することによって、プログラムを作成して行きます。イベントは、画面上のボタンをクリックしたときやキーを押したときなど、ユーザーの操作によって発生する場合と、あらかじめ設定されていた時間が経過したときのように、システム側が発生させる場合があります。

イベント ドリブン

Visual Basic の新しいプログラミング モデルを示す用語です。イベント ドリブンのプログラムでは、それぞれの処理がソース コードに記述された順番に行われるのではなく、イベントの発生によってプロシージャが呼び出され処理が行われます。

イベント プロシージャ

特定のイベントが発生したときの応答として、自動的に呼び出されるプロシージャです。それぞれのイベントに対する処理を、イベント プロシージャとして記述します。イベント プロシージャは値を返さないため、**SUB** プロシージャの一種であると言えます。

ウィンドウ

ある処理を行うための画面上の区切られた領域を、一般にウィンドウと呼びます。フォームやダイアログ ボックスもウィンドウの一種です。

オブジェクト

Visual Basic では、フォーム，コントロール，特殊オブジェクト (**CLIPBOARD, PRINTER, SCREEN**) を総称して、オブジェクトと言います。

オプション ボタン

グループ化されたいくつかの選択肢から 1 つを選ぶためのコントロールです。

関数 (Function)

計算を行って、値を返すコードです。変数と同様に式の中で使われます。Visual Basic が最初から用意している組み込みの関数と、ユーザーが必要に応じて定義するユーザー定義関数があります。

コード モジュール

フォームに関連しないコードだけを含むモジュールです。主にアプリケーションの初期化処理や複数のモジュールが共通に使用するプロシージャや宣言などを記述します。コード モジュールの拡張子は .BAS です。QuickBasic や Microsoft Basic Professional Development System で作成したモジュールは、すべてコード モジュールであることになります。

コマンド ボタン

クリックされたときにコマンドを実行するためのコントロールです。

コントロール

コマンド ボタンやチェック ボックス，メニューなど、アプリケーションの操作に使われるオブジェクトです。コントロールはフォーム上に配置されます。

コンボ ボックス

リスト ボックスとテキスト ボックスを組み合わせたコントロールです。ユーザーは、ドロップダウンするリスト ボックスから 1 つを選択することも、テキストを直接入力することもできます。

垂直、水平スクロール バー

特定の範囲内の位置を指定するコントロールです。スクロールだけでなく、例えばボリュームの調節などにも応用できます。

ステートメント

一般に、プログラムの最小実行単位をステートメントといいます。Visual Basic のステートメントには、入出力や代入などの実際の処理を行う命令 (**PRINT, LINE**)，プログラムの流れを制御する命令 (**IF..THEN..ELSE, DO..WHILE**)，宣言を行う命令 (**DECLARE, DIM**) などがあります。

ダイアログ ボックス

アプリケーション中で、設定や確認のために一時的に表示されるウィンドウを、一般にダイアログ ボックスと呼びます。Visual Basic for MS-DOS では、ダイアログ ボックスを表示するにはいくつかの方法があります。

タイマー

一定の時間間隔でイベントを発生させるためのコントロールです。

チェック ボックス

条件のオン/オフを切り換えるためのコントロールです。

テキスト ボックス

テキストの入力と編集を行うためのコントロールです。カット，コピー，ペーストの機能があらかじめ組み込まれています。

デスクトップ

フォームの背景である、画面全体を指します。フォームはデスクトップの上に表示されます。デスクトップの色やパターンは **SCREEN** オブジェクトの ControlPanel プロパティで設定します。デスクトップ上にフォームが表示されていないときには、**PRINT** や **LINE** などのグラフィックス ステートメントによって図形や文字を書くことができます。グラフィック データはデスクトップには表示できますが、フォーム上に表示することはできません。

特殊オブジェクト

あらかじめ定義されている、画面には表示されないが特別な機能を持っているオブジェクトで、**CLIPBOARD, PRINTER, SCREEN** の 3 種類があります。

ドライブ，ディレクトリ，ファイル リスト ボックス

それぞれ、ドライブ名，ディレクトリ名，ファイル名を一覧表示して選択するためのコントロールです。目的が限定されたリスト ボックスと言うことができます。

ピクチャー ボックス

テキストを表示すると同時に、オプション ボタンなどのグループ化に使うためのコントロールです。

フォーム

アプリケーションの操作の中心となるウィンドウです。この上に、コマンド ボタンやテキスト ボックスなどのコントロールを配置します。フォームは画面上に同時に複数開くことが可能で、他のフォームに重ねて表示することができます。

フォーム モジュール

フォームの外観とフォームに“アタッチ”されているコードを含むモジュールです。フォーム モジュールの拡張子は .FRM です。

フレーム

他のコントロールを機能別にグループ化して配置するためのコントロールです。主にオプション ボタンやチェック ボックスのグループ化に使われます。

プロジェクト

1 つのアプリケーションを構成するファイルの集合です。この中にはフォーム モジュールやコード モジュールが含まれます。ひとつのプロジェクトを複数のフォーム モジュールで構成することも可能です。プロジェクトは、拡張子 .MAK のプロジェクト ファイルによって管理されます。このファイルはテキスト ファイルで、プロジェクトに含まれるすべてのモジュールのファイル名が書かれています。

プロシージャ

Basic の構造化に伴って導入された、プログラム中の処理の単位です。値を返すものは **FUNCTION** プロシージャ、返さないものは **SUB** プロシージャと呼ばれます。

プロパティ

オブジェクトの属性です。フォーム デザイナでオブジェクトのプロパティを設定することによって、特徴 (サイズや色など) や動作 (表示/非表示など) が定義できます。オブジェクトの種類によって、持っているプロパティが異なります。例えばリスト ボックスには、幅 (Width) や高さ (Hight)，色 (BackColor, ForeColor)，マウス ポインタの形 (MousePointer) など、16 種類のプロパティがあります。プロパティは、プログラム中から操作することもでき、現在の設定を調べたり、代入文によって値を変更したりすることが可能です。

メソッド

オブジェクトを操作する専用の命令 (Basic コード) です。オブジェクトの種類によって適用できるメソッドが異なります。いくつかのメソッド (**PRINT** や **CLS**) はステートメントと同じ名前を持っていますが、オブジェクトに対して使われた場合は自動的にメソッドとして扱われ、デスクトップに対して使われる場合はステートメントとして扱われます。プログラム中からオブジェクトを操作するには、メソッドを使う以外に、代入文によってプロパティに値をセットする方法があります。

メニュー

ドロップダウンして表示されるコマンドを選択し、処理を呼び出すためのコントロールです。最大で 5 段階までの階層を持つメニューを設計できます。

モーダルとモードレス

ウィンドウには、モーダルとモードレスの 2 種類があります。アクティブなウィンドウ (フォームやダイアログ ボックス) がモーダルであるときは、了解または取消の動作によってそのウィンドウをクローズするまで、アプリケーションのユーザーは他のウィンドウに切り換えることができません。例えば、プログラミング環境でファイルをロードまたはセーブするときに表示されるダイアログ ボックスはモーダルであり、他のウィンドウやメニューなどをクリックしても、そのダイアログ ボックスがアクティブなままになります。これに対して、アクティブなウィンドウがモードレスであるときは、ユーザーが他のウィンドウに切り換える前に、了解または取消の動作によってクローズする必要はありません。ユーザーは、いったん別のウィンドウに切り換えて作業してから、再びそのウィンドウに戻って作業を続けることができます。

ラベル

実行時にユーザーが変更できないテキストを表示するためのコントロールです。

リスト ボックス

リストから 1 つの項目を選ぶためのコントロールです。

CLIPBOARD オブジェクト

テキストのカット，コピー，ペーストに使われるバッファ領域です。Visual Basic for MS-DOS では特殊オブジェクトとして扱われます。

PRINTER オブジェクト

プリンタなどの出力先を示す特殊オブジェクトです。出力先のプリンタやファイルを決め、出力をコントロールします。

SCREEN オブジェクト

システム全体の色や属性の設定を行うための特殊オブジェクトです。このオブジェクトを操作することによって、画面上のすべてのオブジェクトの色や属性を変更できます。

提供されているツール

Visual Basic for MS-DOS で提供されているツールを次に示します。

VBDOS.EXE

Visual Basic for MS-DOS のプログラミング環境。アプリケーションの編集やデバッグを行います。

FD.EXE

Visual Basic for MS-DOS のフォーム デザイナ。アプリケーションのユーザー インターフェイスの設計に使います。

BC.EXE

Visual Basic for MS-DOS のネイティブコード コンパイラ。LINK.EXE (Microsoft Segmented-Executable Linker) および LIB.EXE (Microsoft Library Manager) と組み合わせて使うことによって、オブジェクト モジュール ライブラリ (.LIB)，Quick ライブラリ (.QLB)，実行可能 (.EXE) ファイルを作成します。

ツールキットとサンプル アプリケーション

Visual Basic for MS-DOS には、次のツールキットとサンプル アプリケーションが付属しています。

コモン ダイアログ ツールキット (CMNDLG*.*)

ファイルのオープン，保管，印刷など、一般的に使われる 7 つのダイアログ ボックスが含まれています。これらのダイアログ ボックスは、プログラム中から簡単に利用することができます。

マウス ツールキット (MOUSE.*)

フォームを使わない プログラムで、マウスをサポートするためのルーチンです。フォームを使うプログラムでは、マウスは標準でサポートされます。

電卓アプリケーション (CALC.MAK)

コマンド ボタン コントロール配列の使い方を示す一般的な電卓プログラムです。キーボードとマウスをサポートしています。

時計アプリケーション (TOKEI.MAK)

デジタル表示のアラーム付き時計です。日付や時刻，表示書式などの関数，タイマー コントロール，メニュー コントロール配列，**INPUTBOX$** および **MSGBOX** 関数の使い方を示しています。

コントロール パネル アプリケーション (CONTROLP.MAK)

システムの色や属性を設定する ControlPanel プロパティの使い方を示すサンプル プログラムです。

ファイル検索アプリケーション (SEEK.MAK)

ファイル リスト ボックス，ディレクトリ リスト ボックス，ドライブ リスト ボックスの使い方を示すプログラムです。ファイルを検索し、ディスク上のファイルの一覧を表示します。

メモ帳アプリケーション (NOTEPAD.MAK)

テキスト ボックス (編集フィールド)，ファイル I/O，**CLIPBOARD** オブジェクト，コモン ダイアログ ツールキットの使い方と、メニュー コントロール配列を使って実行時にメニューを作成する方法を示します。

Quick ライブラリ ビューア (QLBVIEW.MAK)

バイナリ ファイル I/O の使い方と既存の Basic プログラム (QuickBasic の QLBDUMP.BAS) にフォームの機能を追加する方法を示すサンプル プログラムです。特殊オブジェクト **SCREEN** の ControlPanel プロパティの使い方も示されています。

ソート デモ アプリケーション (SORTDEMO.MAK)

ソートの一般的なアルゴリズムを示すプログラムです。

立体グラフィックス アプリケーション (TORUS.MAK)

フォームとグラフィックスを同時に使う方法と、既存の Basic プログラム (QuickBasic の TORUS.BAS) にフォームを使った機能を追加する方法を示すサンプル プログラムです。

グラフ アプリケーション (GRAPHICS.MAK)

フォームとグラフィックスを組み合わせる方法を示すプログラムです。フォームに入力された値に従って棒グラフを描きます。

この製品で提供されている他のすべてのファイルについては、配布ディスク Disk 1 中の PACKING.TXT ファイルを参照してください。

Visual Basic for MS-DOS のマニュアル構成

Visual Basic for MS-DOS のパッケージには、次の 3 冊のマニュアルと、オンライン ヘルプが含まれています。

- 『プログラミング ガイド』(このマニュアル)

  Visual Basic for MS-DOS を使ってどのようにアプリケーションを作成するかを、例をまじえながら説明します。実際に簡単なアプリケーションを作成しながら、プログラミング環境でコードを記述する方法と、フォーム デザイナでフォームを設計，作成する方法を説明した後、Visual Basic for MS-DOS がサポートしている各機能をどのように活用していくかを順に触れて行きます。このマニュアルを読んで、コードの動作と意味について理解してください。

- 「リファレンス」

  Visual Basic for MS-DOS の言語仕様に関するリファレンスです。関数，ステートメント，メソッド，プロパティ，イベント，オブジェクトの詳細について、アルファベット順に説明しています (同じリファレンス情報はオンライン ヘルプにも含まれています)。最初に、Visual Basic がサポートしている関数，ステートメント，メソッド，プロパティ，イベント，オブジェクトを機能別に要約し、それぞれのオブジェクトに適用可能なメソッド，プロパティ，イベントの対応表を示してあります。また付録には、予約語の一覧表，データ型と演算子の解説，トラップ可能なエラー メッセージの一覧などが含まれています。プログラム例は、オンライン ヘルプを参照してください。

- 『機種依存情報』

  グラフィックスやキーボード スキャン コードなどの、ハードウェアに依存する機能について説明しています。例えば、グラフィックスを表示するときに必要なスクリーン モードの種類と値などは、このマニュアルに載っています。『プログラミング ガイド』および『リファレンス』には、ハードウェアに依存した機能の詳細は記載されておらず、このマニュアルを参照するようになっています。

  また、Microsoft の以前のバージョンの Basic との互換性についても、このマニュアルを参照してください。

- オンライン ヘルプ

  Visual Basic for MS-DOS の操作方法と言語仕様についてのリファレンスです。マニュアルを読んで Visual Basic の基本的な内容について理解した後は、構文などの細かい仕様やメニュー コマンドの機能を調べるために、このオンライン ヘルプを利用してください。オンライン ヘルプのトピックを表示するには、キーワードにカーソルを合わせて [F1] キーを押します。また、ダイアログ ボックスなどで <ヘルプ> ボタンがあれば、より詳しい説明を含むトピックが用意されています。ほとんどの関数やステートメントの説明には、プログラム例が含まれます。これらの例は、コード ウィンドウに複写/貼り付けして実行でき、アプリケーションで利用することができます。

このマニュアルの構成

このマニュアルは 3 部構成になっており、それぞれ次のような内容について解説しています。

- 第 1 部「Visual Basic の使い方」では、Visual Basic for MS-DOS を使った基本的なプログラミングの方法について説明しています。フォームのデザインの方法，コントロールやプロパティの機能と意味，イベント ドリブンの基本などについて記述してあります。

- 第 2 部「プログラミング テクニック」では、入力の取得，出力の表示，マウス イベントの処理など、一般的なタスクを実行するコードの書き方について説明します。

- 付録では、メモリ管理，プログラミングのヒント，Visual Basic for MS-DOS のメニューなどについて説明します。

マニュアルの読み進め方

Visual Basic for MS-DOS の使い方についてどの段階から学んだらよいかは、Basic とイベントドリブン プログラミングの経験によって異なります。一般に、Visual Basic for MS-DOS のマニュアルは次のようにお読みになることをおすすめします。

1. まず、このマニュアルの第 1 部を読みます。実際に操作しながら読むと効果的です。Visual Basic for MS-DOS のプログラミングの基本を理解してください。
2. 第 2 部を読み、もっとも頻繁に使うプログラミング テクニックを習得します。
3. 実際にプログラムを作成しながら、必要に応じてヘルプまたは『リファレンス』を参照します。また、ハードウェアに依存する機能については、『機種依存情報』またはオンライン ヘルプを参照してください。

イベントドリブン プログラミングと Basic についてすでにかなりの知識がある方は、次のようにしてください。

1. 第 2 章と第 6 章を読んだ後、Visual Basic for MS-DOS を実際に使ってみてください。必要に応じて、ヘルプまたは『リファレンス』を参照してください。
2. 第 1 部の残りの部分をざっと読みます。
3. 第 2 部を読んで、もっとも一般的なコーディング テクニックを理解します。

さらに高度な内容について知りたいときは、次の付録を参照してください。

- プログラミング環境が使うメモリの管理方法: 付録 B 「メモリ管理」
- 一般的なプログラミングのヒント: 付録 C 「プログラム作成上のヒント」
- デバッグのテクニック: 第 17 章 「アプリケーションのデバッグ」

マニュアルの表記規則

このマニュアルは、次の表記規則に従って記述されています。

表記	説明
SUB, IF, CHDIR, PRINT, TIME$	大文字のボールド (太字) は、Visual Basic for MS-DOS の予約語であることを表します。
BackColor, Click	プロパティとイベントの名前は、大文字で始まります。
setup	プログラマが実際に入力する単語はボールドで示します。
eventname	構文中のイタリック (斜体) は、ファイル名のように適当な文字列に置き換える必要のあるものを表します。単語の最後の文字は、指定するデータ型を表すこともあります。
[*expressionlist*]	構文中の角カッコ ([]) は、囲まれた項目が省略できることを表します。

{While|Until}	構文中の大カッコ ({}) と縦棒の組み合わせは、複数の項目から選択する必要があることを表します。全体が角カッコ ([]) で囲まれていなければ、いずれか 1 つの項目を選ばなければなりません。

`SUB HelloButton_Click ()`
`  Readout.Text = "Hello, world!"`
`END SUB`
	この書体は、変数，コード，例，エラーメッセージを表します。

`Readout.Text = "This should all be` `⇒ on one line in the Code window."`	行連結文字 (⇒) はコードが次の行に続くことを表します。実際にプログラムとして入力するときは、1 行で入力します。
CONSTANT.BI, F1_KEY	ファイル名や定数名は、すべて大文字で示してあります。
[ENTER]	説明文中の角カッコ ([]) は、キーボード上のキーやメニュー項目，ダイアログ ボックスの名前などを表します。 Visual Basic for MS-DOS のマニュアルやオンライン ヘルプでは、キーの表記にジェネリック表記を採用しています。一部のキーは、キーボード上に書かれた表記と異なりますから注意してください。例えばリターン キーは [ENTER]，メニューをアクティブにするためのキーは [メニュー] キーです。 Visual Basic for MS-DOS でのジェネリック表記は、Microsoft Windows 用のアプリケーションと合わせているため、 Microsoft Basic や QuickBasic での表記とは異なります。 ジェネリック表記と実際のキーボード上の表記との対応は、『機種依存情報』またはオンライン ヘルプを参照してください。
[メニュー] + [F1]	プラス記号 (+) は、キーの組み合わせを表します。例えば [メニュー] + [F1] は、[メニュー] キーを押しながら [F1] キーを押すことを表します。
Multiple Document Interface (MDI)	略語はその用語を最初に使うときは正式名称を書きます。

第 1 部

Visual Basic の使い方

1 Visual Basic のインストールと実行

Visual Basic for MS-DOS を使うためには、最初にハード ディスクにインストールする必要があります。セットアップ プログラム (SETUP.EXE) を使ってインストールを行います。パッケージに含まれているディスクのファイルは圧縮されているため、MS-DOS の **COPY** コマンドでファイルをコピーするだけでは使えません。

▼ Visual Basic をインストールするには

1. フロッピー ディスク「Disk 1 (Setup)」をフロッピー ディスク ドライブに挿入します(ご使用の機種によっては、フロッピー ディスクが何種類か用意されている場合があります。ラベルに従って、適切なフロッピー ディスクを選択してください)。
2. 次のように入力します。"B:"の部分はフロッピー ディスクを挿入したドライブ名を入力してください。

   ```
   B:SETUP
   ```

3. 画面の指示に従って、操作を進めます。

セットアップ プログラムのメニューから、関連するドキュメントファイルを表示できます。特に README.TXT ファイルには重要な情報が記述されていますので、必ずお読みください。

すべてのファイルのインストールが完了したら、PATH, LIB, INCLUDE の各環境変数に、Visual Basic をインストールしたディレクトリを追加します。セットアップ プログラムは NEW-VARS.BAT というバッチ ファイルを生成しますので、このファイルを参考に、AUTOEXEC.BAT ファイルを書き換えてください。また、Visual Basic をインストールしたディレクトリに移動して、NEW-VARS.BAT を実行してもかまいません。環境変数および AUTOEXEC.BAT ファイルについては、MS-DOS のマニュアルを参照してください。

セットアップ プログラムを実行すると、次に示すファイルがインストールされます。

- Visual Basic のプログラミング環境本体 (VBDOS.EXE), フォーム デザイナ (FD.EXE)。
- 各種のサンプル プログラム (Tokei, Calc, Notepad など)。
- 各種のユーティリティ。Microsoft Visual Basic for MS-DOS のコンパイラ (BC.EXE), Microsoft ライブラリ マネージャ (LIB.EXE), リンカ (LINK.EXE), Visual Basic for MS-DOS のランタイム モジュール (VBDRT10.EXE) など。

これらに加えて、セットアップ プログラムは、ライブラリ, ドライバ, オブジェクト ファイル, ヘルプ, 各種のテキスト ファイルなどをハード ディスク上にインストールします。「Disk 1 (Setup)」に含まれている PACKING.TXT ファイルに、セットアップ プログラムでインストールされる全ファイルのリストがあるので参考にしてください。

1.1　Visual Basic の起動

▼ Visual Basic を使うには

- MS-DOS のコマンドラインで、**VBDOS** と入力してください。

  Visual Basic の起動画面が表示されたら、[ENTER] キーを押してください。図 1.1 に示すプログラミング環境が表示されます。

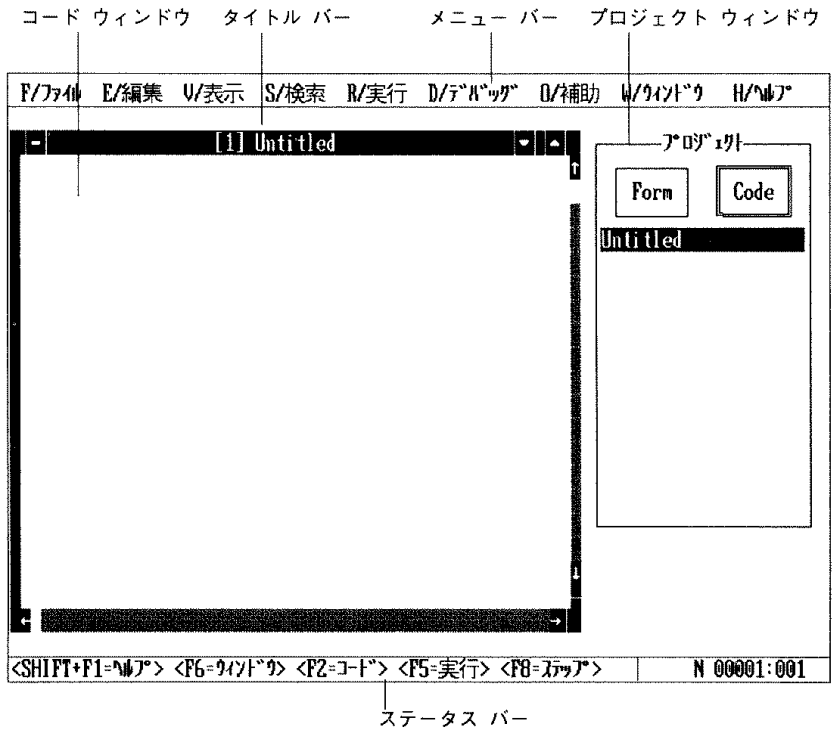


図 1.1　プログラミング環境

Visual Basic を起動するときに、コマンド ラインで各種のオプションを指定することができます。例えば、/L オプションの後にQuick ライブラリ名を指定すると、プログラミング環境に Quick ライブラリがロードされます。コマンドラインで指定可能なオプションについては、『リファレンス』またはオンライン ヘルプを参照してください。

プログラミング環境に表示される主な項目は、次のとおりです。

- メニュー バーには、メニュー名が表示されます。特定のメニューに関するヘルプを表示するには、そのメニューのタイトルが反転表示された状態で [F1] キーを押します。Visual Basic のすべてのメニューとその機能については、付録 D を参照してください。
- コード ウィンドウには、編集対象となっている Basic プログラムのコードが表示されます。
- タイトル バーには、現在アクティブになっているモジュール, プロシージャ, ウィンドウなどの名前が表示されます。
- ステータス バーには、カレントのウィンドウやコマンドに関する、ショートカット キーや簡単なヘルプ情報が表示されます。カーソルのある行と列も、ここに表示されます。
- プロジェクト ウィンドウには、カレントのプロジェクトに含まれるファイルのリストが表示されます。Visual Basic では、これらのファイルを“モジュール”と呼びます。

1.2 Visual Basic の概要

Visual Basic for MS-DOS は、MS-DOS アプリケーションを開発するための、フルセットの環境を提供します。Visual Basic for MS-DOS を使うと、ウィンドウ, ダイアログ ボックス, メニュー, コマンド ボタン, オプション ボタンなどのインターフェイスを備えたアプリケーションを手軽に作成できます。Visual Basic for MS-DOS のプログラミング環境上で記述したプログラム コードは、[SHIFT] + [F5] キーを押すことで、インタープリタによってすぐに実行できます。また、プログラムにエラーがあると、環境内に組み込まれているデバッガがその位置を示すので、その場で修正できます。さらに、コンパイラ, リンカ, ライブラリ マネージャなどのアプリケーション開発ツールを使って、実行可能 (.EXE) ファイルと Quick ライブラリ (.QLB) を作成できます。

Visual Basic は、大きく分けてプログラミング環境とフォーム デザイナの 2 つの部分から構成されています。ソース コードを書いたり、プログラムの実行とデバッグを行うときは、プログラミング環境を使います。そして、アプリケーションのインターフェイス部分の作成や変更を行うときにはフォーム デザイナに切り換えます。フォーム デザイナでの作業が終わったら、再度プログラミング環境に戻ります。

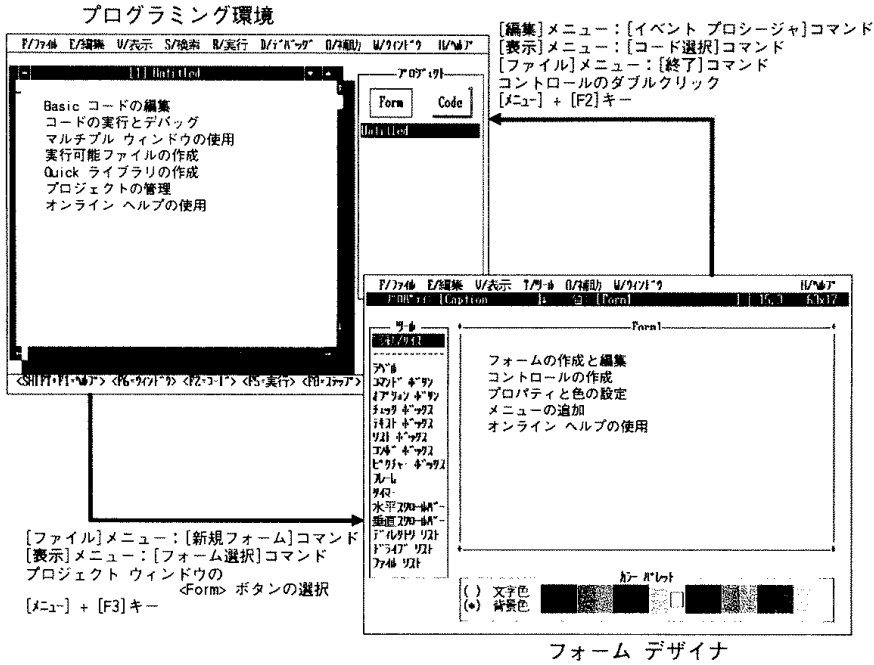


図 1.2　プログラミング環境とフォーム デザイナの切り換え

プログラミング環境とフォーム デザイナの切り換えは、メニュー コマンドによって行います。そのとき行っていた作業の結果をディスクに保管していないときは、ダイアログ ボックスが表示されるので、その指示に従ってください。

▼ プログラミング環境からフォーム デザイナに切り換えるには

- 新しくフォームを作成するときは、[ファイル] メニューから [新規フォーム] を選びます。

すでにフォームを作成している場合は、次のいずれかの方法でフォーム デザイナに切り換えることができます。

- [表示] メニューから [フォーム選択] を選び、表示されたダイアログ ボックスの中でフォームを指定します。
- [メニュー] + [F3] キーを押して、表示されたダイアログ ボックスの中でフォームを指定します。
- プロジェクト ウィンドウの中の <Form> ボタンをクリックします。

▼ フォーム デザイナからプログラミング環境に切り換えるには

次のいずれかの方法でプログラミング環境に戻ることができます。

- [編集] メニューの [イベント プロシージャ] コマンドを選び、表示されるダイアログ ボックスでオブジェクトとイベントを指定します。
- [メニュー] + [F2] キーを押し、表示されるダイアログ ボックスでオブジェクトとイベントを指定します。

- フォーム上のオブジェクトをダブルクリックし、表示されるダイアログ ボックスでオブジェクトとイベントを指定します。
- [表示] メニューの [コード選択] コマンドを選び、表示されるダイアログボックスでプロシージャを選びます。
- [ﾌｧｲﾙ] メニューの [終了] コマンドを選びます。

1.3 Visual Basic のウィンドウ

Visual Basic のプログラミング環境には、次に示す 7 種類のウィンドウがあります。新しいウィンドウを表示したり表示を切り換えたりするには、[ｳｨﾝﾄﾞｳ] メニューのコマンドを使います。

ウィンドウ	目的
コード ウィンドウ	Basic ソース コードの表示と編集を行います。複数のコード ウィンドウを開いて、同じプログラムの別の場所を同時に表示, 編集することも可能です。最大で 9 つまで同時に開くことができます。
デバッグ ウィンドウ	ウォッチおよびウォッチ ポイントの状態を表示します。アプリケーションをデバッグするときに“ウォッチ”機能によって、プログラム中の変数と式の値を調べることができます。また、“ウォッチ ポイント”で設定した条件文で、評価した式の値が偽 (0) から真 (-1) に変わったときにアプリケーションの実行をいったん停止させることができます。
コール ヒストリ ウィンドウ	実行の現在位置に至るまでの、プロシージャが呼び出された流れを表示します。
ヘルプ ウィンドウ	オンライン ヘルプを表示します。
イミディエイト ウィンドウ	Basic コードを 1 行ずつ実行します。
出力画面ウィンドウ	アプリケーションの実行によって画面に出力された結果 (テキスト画面) を表示します。
プロジェクト ウィンドウ	カレント プロジェクトのコード モジュールとフォーム モジュールのリストを示します。

<編集> と <新規ウィンドウ編集>

プログラミング環境のいくつかのダイアログ ボックスには、<編集> と <新規ウィンドウ編集> の 2 つのボタンがあります。新規ウィンドウ編集を選ぶと、新しいコード ウィンドウを開いて、そこに指定したソース コードを表示します。

▼ **そのときアクティブなコード ウィンドウに指定したコードを表示するには**

- <編集> を選びます。このボタンをクリックすると、アクティブ ウィンドウの表示内容が入れ替わりますが、プロジェクトからファイルを削除するわけではありません。

▼ 新しいコード ウィンドウに指定したコードを表示するには

- <新規ウィンドウ編集> を選びます。新しい コード ウィンドウが自動的にアクティブになり、他のコード ウィンドウにオーバーラップして表示されます。

ウィンドウの操作方法

ウィンドウは、次のような構成になっています。

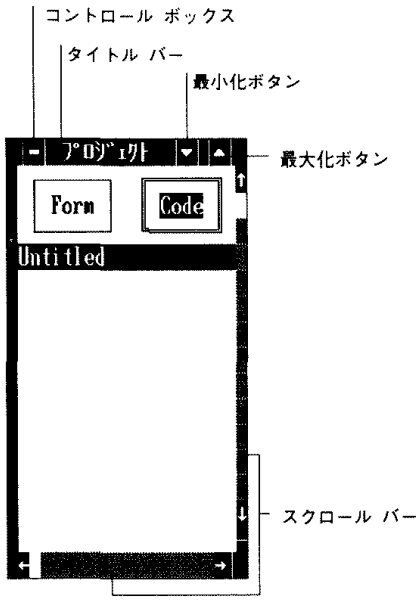


図 1.3　プロジェクト ウィンドウ

それぞれのウィンドウは、サイズ変更, 最小化 (アイコン化する), 復元 (元の大きさに戻す), 最大化 (画面いっぱいに拡大する), クローズの操作が可能です。ウィンドウに対する共通操作を、表 1.1 に示します。

表 1.1　ウィンドウに対する操作

操作内容	操作方法
ウィンドウのサイズを変更する	マウスでウィンドウの境界をドラッグします。
ウィンドウを移動する	マウスでタイトル バーをドラッグします。
ウィンドウを最小化する	ウィンドウ右上の最小化ボタン (下矢印) をクリックします。ウィンドウはアイコンに最小化され、画面の一番下に表示されます。 アイコンの上に他のウィンドウを重ねてもかまいません。見えなくなったアイコンは、画面の一番下の部分をおおっているウィンドウのサイズを変えるか、動かすことによって再び現れます。

最小化したウィンドウを元の大きさに戻す	アイコンをダブルクリックします。
ウィンドウを最大化する	ウィンドウ右上の最大化ボタン (上矢印) をクリックするか、ウィンドウのタイトル バーをダブル クリックします。ウィンドウが最大化され、画面いっぱいの大きさになります。また、最大化ボタンは復元ボタン (上下矢印かダイヤモンド マーク) に変ります。
最大化したウィンドウを元の大きさに戻す	ウィンドウ右上の復元ボタン (上下矢印かダイヤモンド マーク) をクリックするか、ウィンドウのタイトル バーをダブル クリックします。
ウィンドウをクローズする	ウィンドウ左上のコントロール メニュー ボックスをダブル クリックします。

これらの操作は、キーボードからでも行うことができます。キーボードからの操作方法については、このマニュアルの裏表紙を参照してください。また、オンライン ヘルプにより詳しい説明があります。[ﾒﾆｭｰ] + [-] キーを押すとアクティブ ウィンドウの左上にコントロール メニューが表示されるので、行いたい操作に対応したコマンドを反転表示させて [ENTER] キーを押してください。ウィンドウの移動やサイズ変更も方向キーを使って行うことができます。

プロジェクト ウィンドウ

プロジェクト ウィンドウ (図 1.3) には、作成しているアプリケーションを構成するファイルの一覧が表示されます。プロジェクト ウィンドウには、<Form> ボタンと <Code> ボタンがあります。編集したいファイルを反転表示して、フォームを編集するときは <Form> ボタン、コードを編集するときは <Code> ボタンを押してください。

ウィンドウの選択

画面上のウィンドウで、操作ができる状態になっているものを“アクティブなウィンドウ”といいます。同時に複数のウィンドウをアクティブにすることはできません。アクティブなウィンドウには、スクロール バー, コントロール ボックス, 最小化および最大化ボタンが表示されます。[ｳｨﾝﾄﾞｳ] メニューでは、カレントのアクティブ ウィンドウ名にアスタリスク (*) が表示されて、他のウィンドウと区別されます。

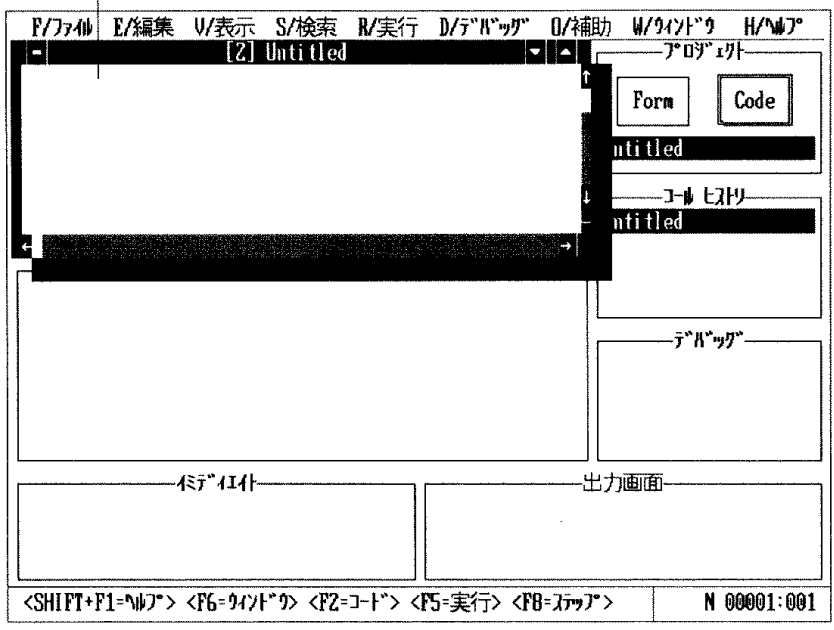


図 1.4　アクティブなウィンドウ

▼ **ウィンドウを選ぶには**

- ウィンドウをクリックします。

  または、

  [ｳｨﾝﾄﾞｳ] メニューから、ウィンドウ名を選択します。

  または、

  [F6] キーを押すと、現在オープンしているウィンドウを順番にアクティブ ウィンドウにすることができます。

1.4　フォーム デザイナの起動

フォーム デザイナでは、アプリケーションのユーザー インターフェイスを設計します。プログラミング環境で、新しくフォームを作成したり、既存のフォームを編集するように指示すると、自動的にプログラミング環境の実行が終了して、フォーム デザイナが起動されます。

▼ **新しいフォームを作成するには**

- [ﾌｧｲﾙ] メニューから [新規フォーム] コマンドを選びます。

▼ **既存のフォームを編集するには**

1. [表示] メニューから [フォーム] を選びます (または [ﾒﾆｭｰ]+[F3] キーを押します)。

   そのときメモリにロードされているフォームのリストが、[フォーム選択] ダイアログ ボックスに表示されます。

2. 編集したいフォーム名を選択します。

3. <了解> をクリックします。

   フォーム デザイナ (図 1.5) が表示されます。

プロジェクト ウィンドウでフォームを選択し、<Form> ボタンをクリックすることによってフォーム デザイナを起動することもできます。

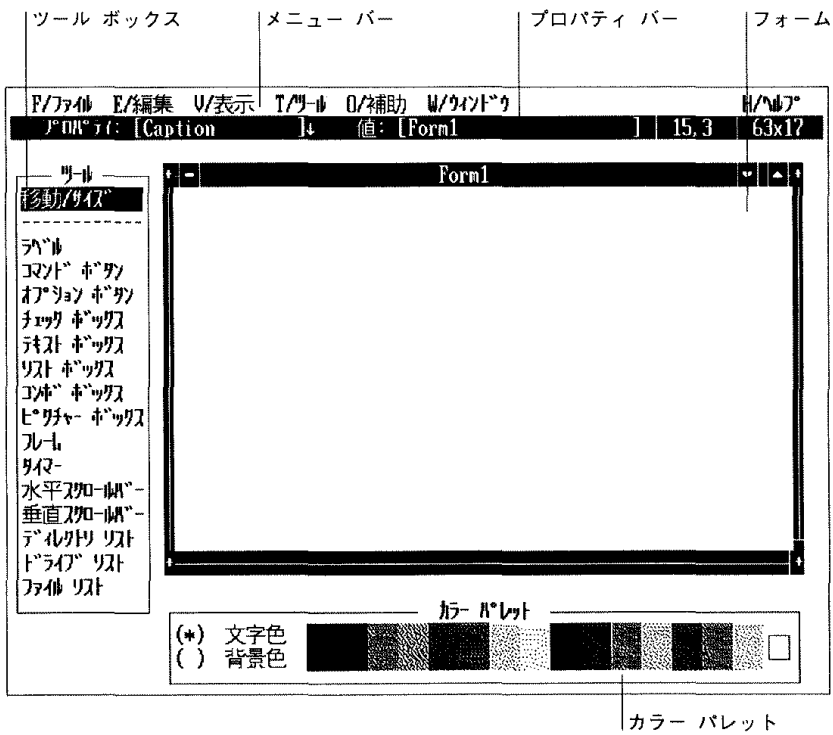


図 1.5 フォーム デザイナ

参 考 コマンド ラインから **FD** と入力しても、フォーム デザイナ (FD.EXE) を起動できます。フォーム デザイナは単独のユーティリティとしても利用できるため、フォーム モジュールをコードと切り離して作成できます。詳しくは、第 6 章「プロジェクトの管理」を参照してください。

フォーム デザイナの画面の構成

フォーム デザイナの画面は次のような部分から構成されています。

名称	説明
フォーム	作成するアプリケーションのユーザー インターフェイスとなるウィンドウやダイアログ ボックスです。
メニュー バー	メニュー名を表示します。
プロパティ バー	フォームまたはコントロールのプロパティの表示と設定を行います。
ツールボックス	フォーム上にコントロールを配置するときに使います。
カラー パレット	フォームまたはコントロールの文字色 (ForeColor) または背景色 (BackColor) を変更します。
メニュー デザイン ウィンドウ	アプリケーションにメニュー バーとメニュー コマンドを追加するときに使います。

プログラミング環境へ戻る

フォームの設計が完了したら、フォーム デザイナを終了してプログラミング環境に戻り、Basic コードの記述とデバッグを行います。

▼ **フォーム デザイナを終了するには**

次のいずれかの操作を行います。

- [ﾌｧｲﾙ] メニューから [終了] コマンドを選びます。
- [表示] メニューから [コード] コマンドを選びます。
- [編集] メニューから [イベント プロシージャ] コマンドを選びます (または [ﾒﾆｭｰ] + [F2] キーを押します)。
- オブジェクト (フォームまたはコントロール) をダブルクリックします。

これらのいずれかの操作を行うと、終了の確認メッセージと、プロジェクトおよびソース ファイルを保存してよいかどうか確認を求めるメッセージが表示された後、プログラミング環境に戻ります。このときプログラミング環境は、フォーム デザイナを呼び出す直前と同じ状態になっています。

1.5　Visual Basic Help の利用

Visual Basic は、必要な情報を簡単に検索できるオンライン ヘルプ (Visual Basic Help) を用意しています。オンライン ヘルプには、次のような説明が含まれています。

- Basic の関数, ステートメント, メソッド, 演算子, キーワード, メタコマンド, ルーチンの解説とプログラム例
- コントロール, 特殊オブジェクト, イベント, プロパティの解説とプログラム例

- メニュー コマンドやダイアログ ボックスの説明
- エラーの原因と対策

さらに、スコープの規約やコントロール配列の作成方法など、さまざまなルールや機能に関する説明も含まれています。

ヘルプの表示方法

Visual Basic Help は、プログラミング環境とフォーム デザイナの両方から利用できます。アクティブなウィンドウに表示されている中で、ヘルプ情報を見たい単語にカーソルを合わせて、[F1] キーを押してください。また、メニュー コマンドのヘルプ情報を見たいときは、メニュー名やコマンドが反転表示されている状態で [F1] キーを押してください。

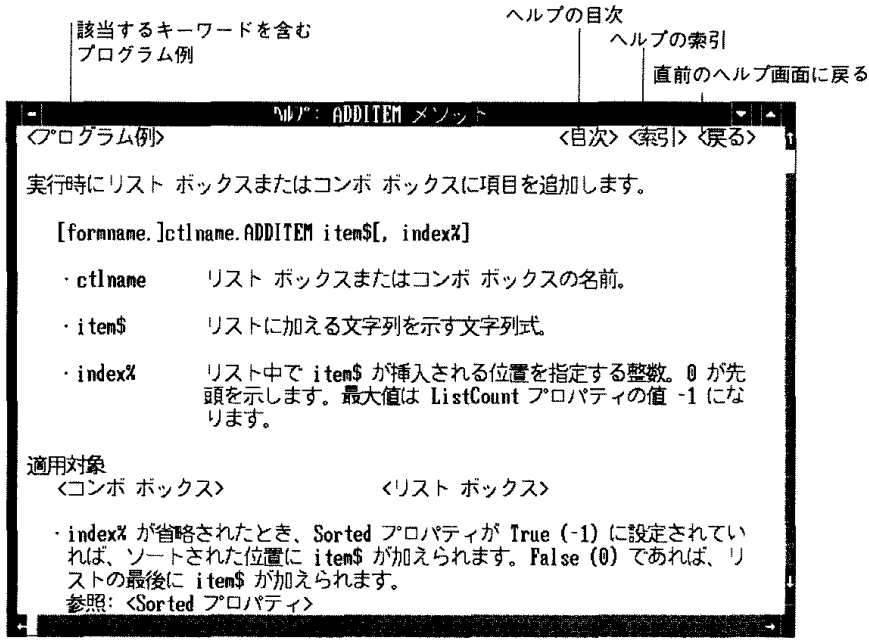
[ヘルプ] メニューには、Visual Basic Help の目次や索引を表示するためのコマンドが用意されています。

ハイパーリンクを使った関連するトピックの表示

ヘルプ ウィンドウの内部では、“ハイパーリンク” を使って、関連する用語のヘルプ情報を表示することもできます。ハイパーリンクが設定されているキーワードや単語は、次の図のように、<> で囲んで示されています (通常、<> は緑色で示されます)。ここをダブルクリックすると関連する用語のヘルプ情報へジャンプできます。また、ヘルプ ウィンドウで [TAB] キーを押すと、ハイパーリンクが設定されている場所にカーソルが順に移動します。

< ヘルプの使い方 > —— ハイパーリンク

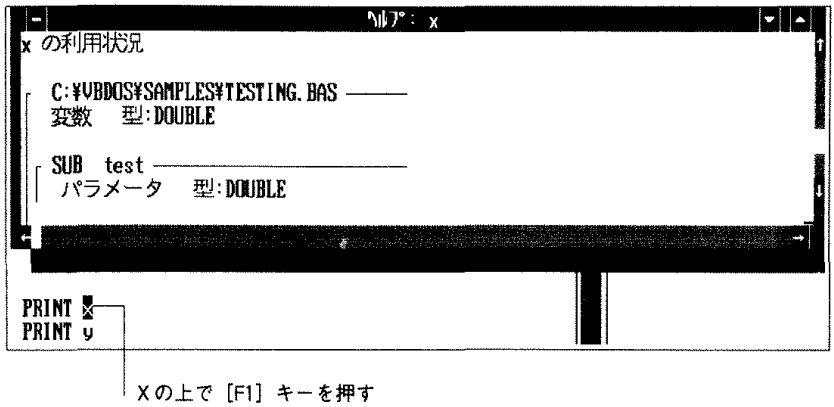
ヘルプ画面の表示例を、次に示します。



その他のヘルプの表示方法

Visual Basic Help を表示するには、他にも次のような方法があります。

- メニュー コマンドに関するヘルプを表示するには、メニュー名やコマンドが反転表示されている状態で、[F1] キーを押します。
- ダイアログ ボックスに関するヘルプを表示するには、ダイアログ ボックス内の <ヘルプ> ボタンを押すか、ダイアログ ボックスが表示されているときに [F1] キーを押します。
- エラー メッセージに関するヘルプを表示するには、エラーに関するダイアログ ボックス内の <ヘルプ> ボタンを押すか、ダイアログ ボックスが表示されているときに [F1] キーを押します。
- Visual Basic のキーワードの一覧を表示するには、ヘルプの目次の画面から <機能別キーワード一覧> を選びます。
- プログラム中の変数やプロシージャに関する情報を表示するには、目的の変数やプロシージャ名にカーソルを合わせて、[F1] キーを押します。



▼ ヘルプ ウィンドウを閉じるには

次のいずれかの方法でヘルプ ウィンドウを閉じることができます。

- [ESC] キーを押します。
- コントロール メニューから、[クローズ] コマンドを選びます。
- コントロール ボックスをダブルクリックします。

マウスの右ボタンを使ったヘルプの表示

[F1] キーと同様に、マウスの右ボタンのクリックでもヘルプを表示することができます。マウスの右ボタンの機能は、[補助] メニューの [マウス設定] コマンドで変更できます。

ヘルプのプログラム例のカットとペースト

ヘルプ ウィンドウに表示されるすべてのテキストは、複写, 貼り付けの編集操作によってコード ウィンドウにコピーすることができます。

▼ ヘルプ画面からの複写と貼り付けを行うには

1. ヘルプ画面上で、複写したいテキストを選択します。マウスの場合は先頭から最後までドラッグします。キーボードで操作する場合は、選択したい範囲の先頭にカーソルを移動して、[SHIFT] キーを押しながらテキストの末尾までカーソルを動かします。
2. [編集] メニューの [複写] コマンドを選びます。
3. コード ウィンドウ内でテキストを挿入したい位置にカーソルを移動して、[編集] メニューの [貼り付け] コマンドを選びます。

1.6 Visual Basic の利用法を学ぶ

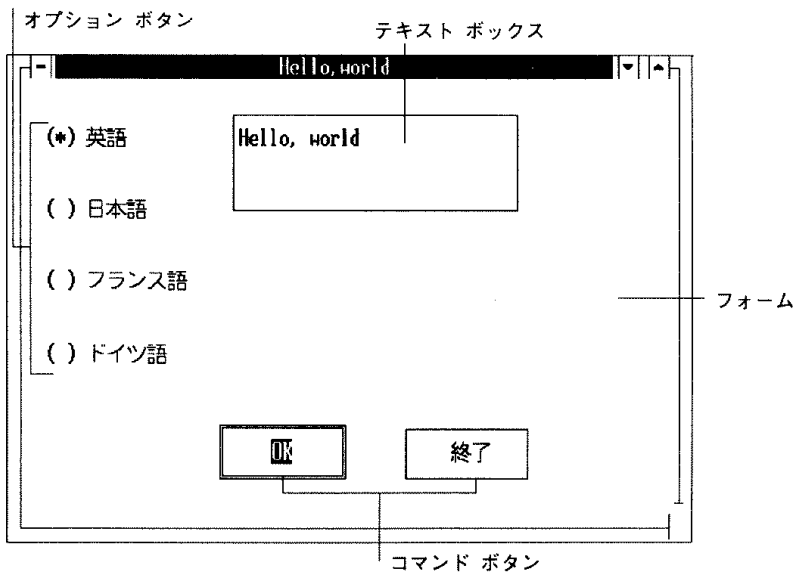
Visual Basic の使い方を学ぶには、次のような方法があります。

項目	説明
プログラミング ガイド (このマニュアル)	第 1 部では、Visual Basic を使ってアプリケーションを作成する方法について説明します。フォーム , コントロール, イベント プロシージャ, プロパティ, Basic 言語の使い方が解説されています。 第 2 部では、デバッグ, グラフィックスの利用, メモリ管理などのプログラミングのテクニックについて説明します。 Visual Basic 全体の用語についての説明も、このマニュアルにあります。
Visual Basic Help (オンライン ヘルプ)	Visual Basic のすべてのリファレンス情報が含まれています。ステートメントや関数の構文, 使用上の注意, ヒント, プログラム例や、機能別に分類されたキーワードやオブジェクトのリストなどがあります。 また、Visual Basic を使ったアプリケーションの作成方法 , エラー メッセージの説明, プログラミング環境とフォームデザイナのメニューやコマンドの説明も用意されています。
リファレンス	Basic 言語に関するアルファベット順のリファレンス , 機能別に分類された言語仕様の解説などが記載されています。

2 Hello, world アプリケーションの作成

通常、Visual Basic のアプリケーションは、フォーム モジュール (.FRM) とコード モジュール (.BAS) の 2 種類のファイルから構成されます。“フォーム” は、Visual Basic アプリケーションの基本的な構成要素となるウィンドウやダイアログ ボックスを示し、入力を得たり出力を表示したりする “コントロール” を含むことができます。フォームとコントロールは、どちらも “オブジェクト” の一種です。

この章では、コマンド ボタンをクリックしたときにテキスト ボックスに "Hello, world" と表示する、簡単なアプリケーションを作成してみます。ここでは、操作手順をマウスの利用を前提として説明します。キーボードだけで作業を行うときは、オンラインヘルプにあるキーボード操作一覧を参照してください。



▼ **このアプリケーションを作成するには**

ここで説明するアプリケーション HELLO.EXE は、次の手順で作成します。

1. フォームを含むプロジェクトを作成します。
2. コマンド ボタン, テキスト ボックス, オプション ボタンをフォームに配置します。
3. 各オブジェクトのプロパティを設定します。
4. アプリケーションの動作を指示する Basic のソースコードを記述します。

5. アプリケーションを保管し、実行します。
6. EXEファイルを作成します。

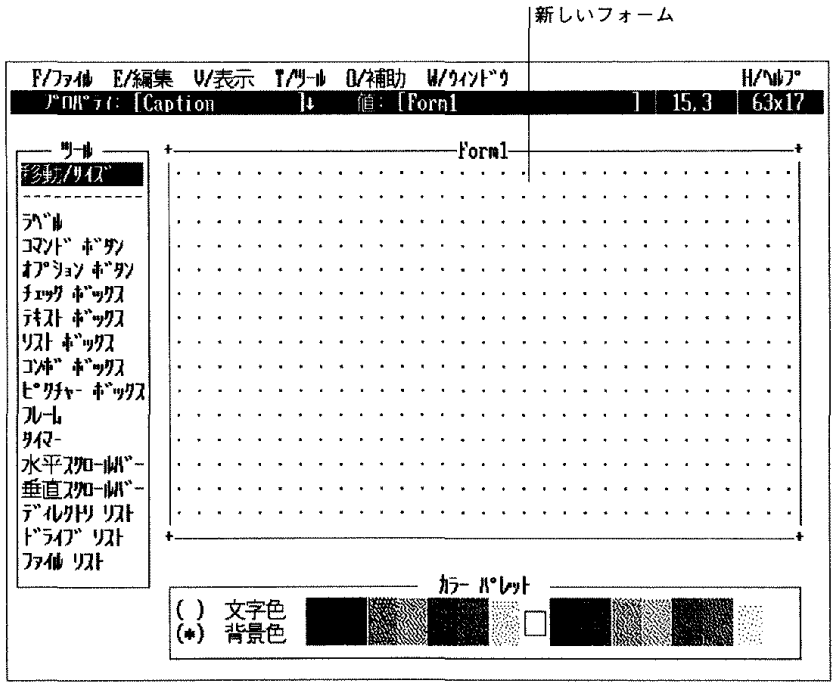
2.1 ステップ 1: プロジェクトとフォームの作成

Visual Basic は、各アプリケーションを、独立した個別のプロジェクトとして取り扱います。“プロジェクト”とは、そのアプリケーションを構成するすべてのソース ファイルの集まりを指します。“Hello, world” アプリケーションの例のようにモジュールが 1 つしかないときでも、独立した 1 つのプロジェクトとして取り扱われます。

▼ **フォームを含む新しいプロジェクトを作成するには**

1. コマンド ラインから **VBDOS** と入力して、Visual Basic を起動します。
2. [ﾌｧｲﾙ] メニューから [新規フォーム] コマンドを選びます。表示される [ファイル名] テキスト ボックスに **HELLO.FRM** と入力して、<了解> をクリックします。

   Visual Basic はフォーム デザイナを起動して、新しいフォームを生成します。



2.2 ステップ 2: フォームへのコントロールの配置

フォームへのコントロールの配置は、作図を行うためのアプリケーションで図形を描くときとよく似た手順で行います。Hello, world アプリケーションにテキスト ボックス，コマンド ボタン，オプション ボタンを配置するには、ツールボックスまたは [ツール] メニューを使います。ツールボックスや [ツール] メニューには、フォームに追加することができる各種のコントロールが含まれています。

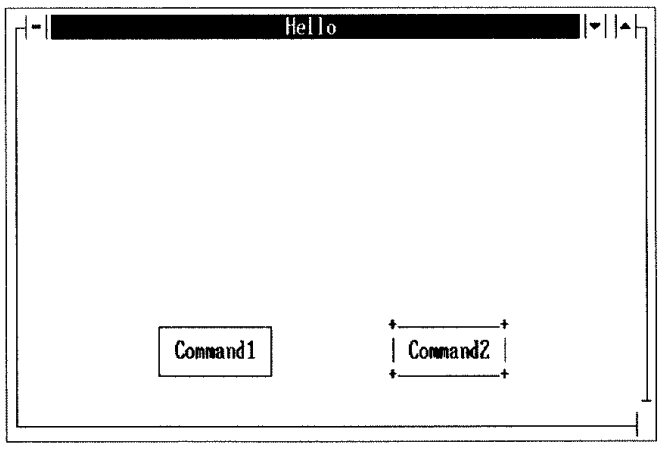
▼ フォームにコントロールを配置するには

1. まずツールボックスの [ｺﾏﾝﾄﾞ ﾎﾞﾀﾝ] をクリックして選択します。次にフォーム上でマウスをドラッグしてコマンド ボタンを配置します。“ドラッグ” とは、マウスの左ボタンを押したままマウスを移動して、ボタンを離す操作のことです。

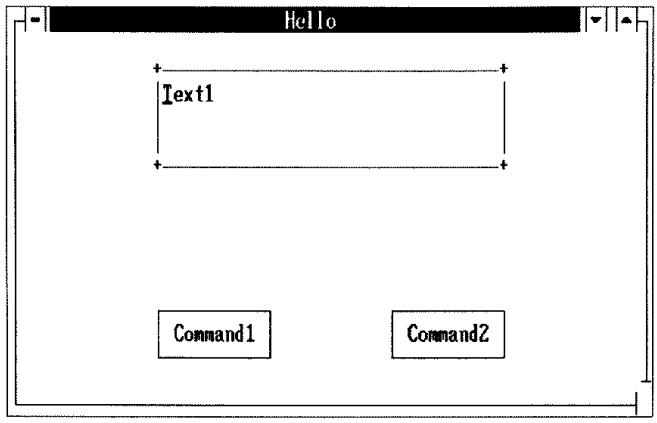
   書き込まれたコマンド ボタンには、デフォルトで Command1 と表示されます。

2. 同じ手順を繰り返して、Command1 の右隣に 2 番目のコマンド ボタンを書き込みます。

   2 番目のコマンド ボタンには、デフォルトで Command2 と表示されます。

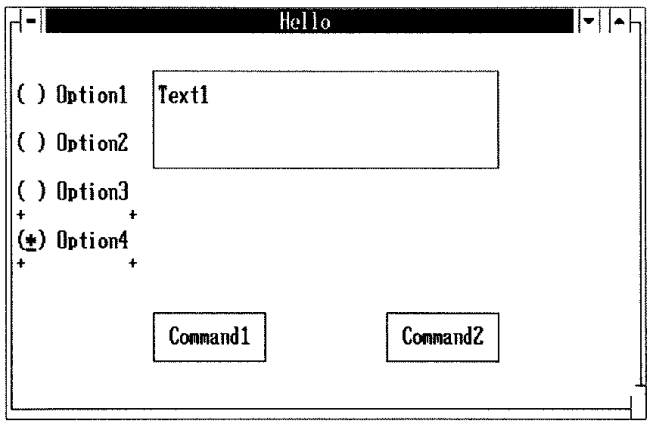


3. ツールボックスの [ﾃｷｽﾄ ﾎﾞｯｸｽ] をクリックして選択し、フォーム上でマウスをドラッグしてテキスト ボックスを配置します。



4. ツールボックスの [ｵﾌﾟｼｮﾝﾎﾞﾀﾝ] をクリックして選択し、フォーム上でマウスをドラッグしてオプション ボタンを配置します。

5. 同じ手順を繰り返して、残りの 3 つのオプション ボタンも配置します。



フォーム上にコントロールを配置するには、ツールボックス上のコントロールの名前をダブルクリックする方法もあります。コントロール名をダブルクリックすると、フォームの中央にデフォルトの大きさのコントロールが表示されるので、必要に応じてそのサイズを変更したり、位置を変えたりしてください。

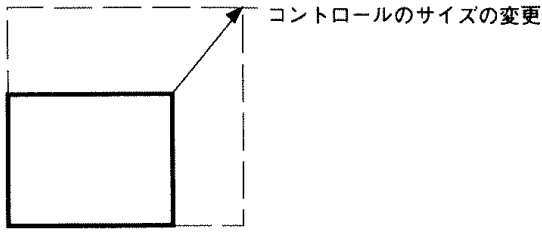
▼ コントロールのサイズを変更するには

1. サイズを変更したいコントロールをマウスでクリックして選択します。

   コントロールの四隅に、“サイズ ハンドル”と呼ばれる“＋”のキャラクタが表示されます。

2. マウス ポインタをサイズ ハンドルに合わせて、コントロールが希望する大きさになるまでドラッグします。

3. マウス ボタンを離します。



▼ コントロールを移動するには

- マウス ポインタを、移動したいコントロール上のサイズ ハンドル以外の位置に合わせて、新しい位置までドラッグします。

2.3 ステップ 3: プロパティの設定

Visual Basic では、それぞれのオブジェクトの特性を、プロパティと呼ばれる属性によって指定します。この特性には、オブジェクト名、内容、サイズ、位置、色などが含まれます。オブジェクトの種類によって、指定可能なプロパティが異なります。新しく作成されたオブジェクトには、自動的にデフォルトのプロパティ値が設定されます。

ほとんどのプロパティは、設計段階 (デザイン時) か実行段階 (実行時) にかかわらず、いつでも設定できます。“デザイン時” とは、フォーム デザイナを使ってアプリケーションのユーザー インターフェイスを作成している段階のことです。デザイン時にプロパティを設定するには、フォーム デザイナのプロパティ バーを使います。

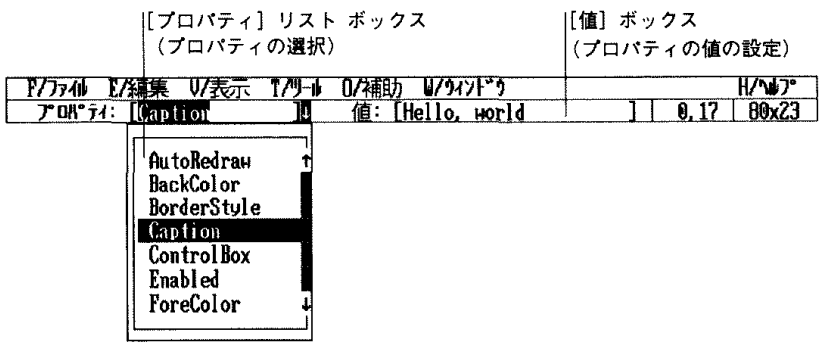


図 2.1 プロパティ バー

“実行時” とは、アプリケーションが実際に実行されている段階を指します。実行時のプロパティの設定は、Basic プログラム (コード) から、変数に値を代入するようにして制御します。

Hello, world アプリケーションで使用するプロパティは、次のとおりです。

プロパティ	内容
Caption	コマンド ボタンやタイトル バーの上や、オプション ボタンの右に表示するテキストを指定します。
CtlName	コード内で参照するための、コントロールの名前を示します。
FormName	コード内で参照するための、フォームの名前を示します
Value	オプション ボタンが選択されたなどの状態を設定します。
Text	テキスト ボックスに表示されるテキストを設定します。

Hello, world アプリケーションにおける、各オブジェクトに対する主なプロパティの設定は次のとおりです。デフォルトから変更したプロパティは、太字で示してあります。

オブジェクト	プロパティの設定
フォーム	**Caption = Hello, world** FormName = Hello
1 つ目のコマンド ボタン	**Caption = OK** CtlName = Command1
2 つ目のコマンド ボタン	**Caption = 終了** CtlName = Command2
1 つ目のオプション ボタン	**Caption = 英語** CtlName = Option1 Value = True
2 つ目のオプション ボタン	**Caption = 日本語** CtlName = Option2 Value = False
3 つ目のオプション ボタン	**Caption = フランス語** CtlName = Option3 Value = False
4 つ目のオプション ボタン	**Caption = ドイツ語** CtlName = Option4 Value = False
テキスト ボックス	**Text** = (なし) CtlName = Text1

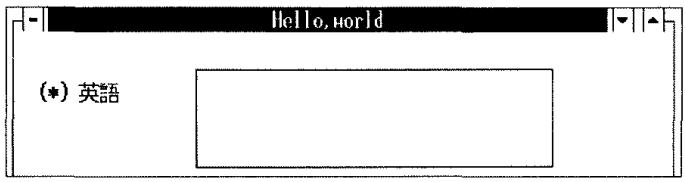
▼ フォームの Caption プロパティを設定するには

1. フォームのタイトル バーをクリックして、フォームを選択状態にします。
2. 画面上の [プロパティ] リスト ボックスから Caption を選びます (図 2.1 を参照してください)。右側にある下向き矢印をクリックすると、リスト ボックスがドロップダウンします。また、[C] キーを押すと、C で始まる名前のプロパティが順に選択されます。

   デフォルトの設定である Hello が [値] ボックスに表示されます。

3. [値] ボックスに **Hello, world** と (ダブル クォーテーションで囲まずに) 入力して [ENTER] キーを押します。

   この時点で、フォームのタイトル バーには Hello, world と表示されます。



▼ **コントロールのプロパティを設定するには**

1. 目的のコントロールを選んでクリックします。
2. [プロパティ] リスト ボックスから設定するプロパティを選びます。
3. [値] ボックスに、設定したい文字列を入力するか、右側にある下向き矢印をクリックしてドロップダウンするリストボックスから設定を選んで、[ENTER] キーを押します。

2.4 ステップ 4: コードの入力

アプリケーション上でのコマンド ボタンの機能は、イベント プロシージャと呼ばれるコードで記述されます。“イベント プロシージャ” は、あるイベントが発生したとき (例えばユーザーが Hello, world フォームの <OK> ボタンをクリックしたとき) に実行されます。

アプリケーションを設計するときは、それぞれのイベントに対してアプリケーションがどのように応答するべきかを書き出してみるとよいでしょう。これによって、どのイベントに対してプロシージャが必要かを明確にすることができます。

Hello, world アプリケーションでは、次の 2 つのイベントに対して応答する必要があります。

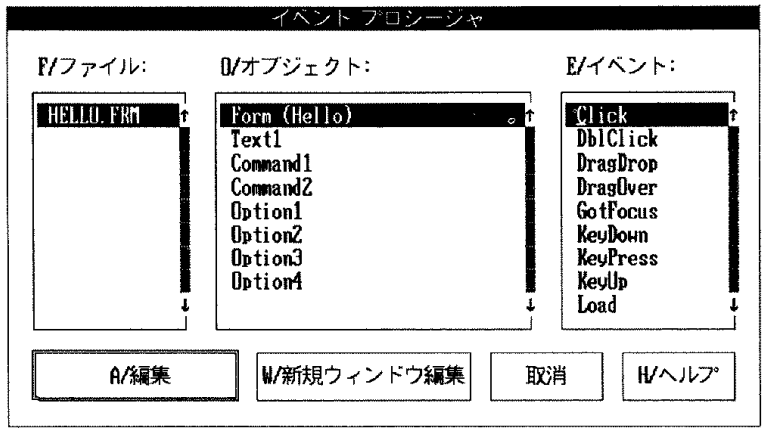
イベント	アプリケーションの応答	イベント プロシージャ
ユーザーが <OK> ボタンをクリックする	テキスト ボックスにメッセージを表示する	Command1_Click
ユーザーが <終了> ボタンをクリックする	アプリケーションを終了する	Command2_Click

▼ **Basic のコードを記述するには**

1. [ﾌｧｲﾙ] メニューから [プロジェクト保管] コマンドを選びます。
2. [プロジェクト保管] ダイアログ ボックスが表示されたら、[ファイル名] テキスト ボックスに **HELLO.MAK** と入力して、[ENTER] キーを押します。
3. [編集] メニューから [イベント プロシージャ] コマンドを選びます。

プログラミング環境に戻るかどうか確認するメッセージが表示されるので、<確認>を選んでください。

画面がプログラミング環境に切り替わり、[イベント プロシージャ] ダイアログ ボックスが表示されます。



フォーム デザイナでフォームに配置されたすべてのオブジェクトとフォーム自身が、[オブジェクト] リスト ボックスに表示されます。これらのオブジェクトからひとつを選ぶと、そのオブジェクトに対して有効なイベントの一覧が [イベント] リスト ボックスに表示されます。

4. [オブジェクト] リスト ボックスから Command1 を選び、[イベント] リスト ボックスから Click を選びます。

5. <編集> ボタンをクリックします。

   コード ウィンドウにCommand1_Click という名前のイベント プロシージャが表示されます。

```
SUB Command1_Click()

END SUB
```

   SUB と **END SUB** ステートメントは、それぞれプロシージャの先頭と最後を表します。このように、イベント プロシージャを新規に作成すると、**SUB** と **END SUB** の 2 つのステートメントが自動的に生成されます。これを、テンプレートと呼びます。

6. **SUB** と **END SUB** の 2 つのステートメントの間に、次に示すコードを入力します。

```
SUB Command1_Click()
    TRUE = -1
    IF Option1.Value = TRUE THEN
        Text1.text = "Hello, world"
    ELSEIF Option2.Value = TRUE THEN
        Text1.text = "世界のみなさん、こんにちは"
    ELSEIF Option3.Value = TRUE THEN
        Text1.text = "Bonjour tout le monde"
    ELSE
```

```
        Text1.text = "Hallo, welt"
    ENDIF
END SUB
```

7. [編集] メニューから [イベント プロシージャ] コマンドを選びます。
8. [オブジェクト] リスト ボックスから Command2 を選び、[イベント] リスト ボックスから Click を選びます。
9. <新規ウィンドウ編集> ボタンをクリックします。

   新しいコード ウィンドウが開いて、Command2_Click という名前のプロシージャが表示されます。

10. **SUB** と **END SUB** の 2 つのステートメントの間に、次のように入力します。

```
SUB Command2_Click()
    SCREEN.HIDE
    END
END SUB
```

2.5 ステップ 5: アプリケーションの保管と実行

▼ **アプリケーションを保管および実行するには**

1. [ﾌｧｲﾙ] メニューから [プロジェクト保管] コマンドを選びます。
2. [実行] メニューから [スタート] コマンドを選びます (または [SHIFT] + [F5] キーを押します)。

   プログラムの入力にまちがいがなければ、Hello, world フォームが画面に表示されます。入力ミスがあると、その場所が強調表示されてエラーメッセージが表示されます。修正して、再度保管してください。

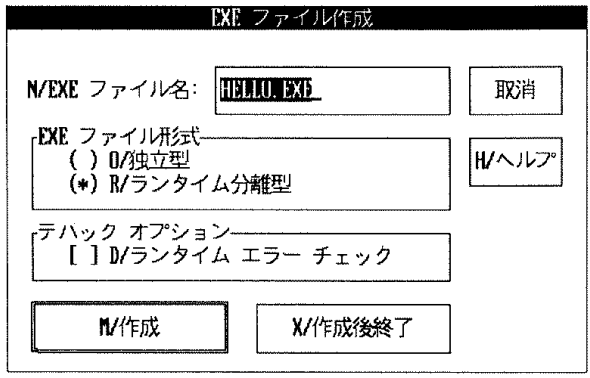
フォーム上のオプション ボタンを選んで OK をクリックし、正しいメッセージがテキスト ボックスに表示されていることを確認してください。アプリケーションを終了するには、<終了> ボタンをクリックします。

2.6 実行ファイルの作成方法

▼ **実行ファイルを作成するには**

1. [実行] メニューから [EXE ファイル作成] コマンドを選びます。

   [EXE ファイル名] ダイアログ ボックスに HELLO.EXE というファイル名が表示されます。



2. <作成後終了> ボタンを選びます。

   カレント ディレクトリ上で HELLO.EXE をコンパイル, リンクし、Visual Basic が終了します。

コマンドラインから HELLO と入力すると、Visual Basic で作成されたアプリケーション HELLO.EXE が起動します。

2.7　アプリケーションの修正

Hello, world アプリケーションの作成とテストが完了したら、これを改良して、ユーザーが <終了> ボタンを押したときに "終了しますか?" というメッセージが表示されるようにしてみます。この機能を実現するには、**MSGBOX** 関数を使うのがもっとも簡単です。

▼ Hello, world アプリケーションを改造するには

1. コマンド ラインから **VBDOS hello** と入力します。この操作で、Visual Basic を起動すると同時に HELLO.FRM が読み込まれます。

2. [表示] メニューから [コード選択] コマンドを選びます (または [F2] キーを押します)。

3. Command2_Click イベント プロシージャを選び、<新規ウィンドウ編集> ボタンをクリックします。

   新しいウィンドウが開いて Command2_Click イベント プロシージャが表示されます。

4. 次のように編集します。

```
x% = MSGBOX("終了しますか？", 1)
IF x% = 1 THEN
    SCREEN.HIDE
    END
END IF
```

MSGBOX 関数は、どのボタンが押されたかに応じて値を返します。x% = 1 であれば、<了解> ボタンが押されたことを示し、アプリケーションは終了します。

MSGBOX 関数について詳しく知りたい場合は、コード ウィンドウで **MSGBOX** 関数を使っている位置にカーソルを合わせ、[F1] キーを押してください。また、ダイアログ ボックスの作成方法に関しては、第 5 章を参照してください。

3 イベントドリブン アプリケーションの作成

Visual Basic では、開発するアプリケーションを、プロジェクトという単位で管理します。1 つのプロジェクトは、複数のフォーム モジュール (.FRM) とコード モジュール (.BAS) を含むことができます。アプリケーションを構成するコードは、次のように分類することができます。

オブジェクトに関連付けられているかどうかでモジュール (ソース ファイル) を分類すると、フォーム モジュールとコード モジュールに分けることができます。

- **フォーム モジュール**

  アプリケーションのユーザー インターフェイスの視覚的構成要素と、そのモジュールのオブジェクトに関連付けられた Basic コードから構成されるソース ファイルです。

- **コード モジュール**

  オブジェクトに直接関連しない Basic のコードだけから構成されるソース ファイルです。

モジュール中でコードが記述されている場所から Basic コードを分類すると、モジュールレベル コードとプロシージャレベル コードに分けることができます。

- **モジュールレベル コード**

  プロシージャの外の部分をモジュールレベルと呼び、そこに記述されたコードをモジュールレベル コードと呼びます。フォーム モジュールのモジュール レベルには、定数や変数, プロシージャの宣言を記述できます。実行文 (実行可能ステートメント) を記述することはできません。コード モジュールのモジュール レベルには、このような制限はありません。

- **プロシージャ レベル コード**

  プロシージャの中をプロシージャ レベルと呼び、そこに記述されたコードをプロシージャレベル コードと呼びます。プロシージャには、値を返す **FUNCTION** プロシージャと、値を返さない **SUB** プロシージャの 2 種類があります。

さらに、イベントに関連付けられているかどうかでプロシージャを分類すると、イベント プロシージャとジェネラル プロシージャに分けることができます。

■　イベント プロシージャ

ユーザーの操作やシステムによって発生するイベントに対応して実行されるプロシージャで、フォーム モジュール内に記述されます。イベント プロシージャは必ず、値を返さない **SUB** プロシージャです。イベントが発生すると、そのイベントに対応するイベント プロシージャが自動的に呼び出されます。イベント プロシージャは **SUB** プロシージャの一種であるため、プログラム中から明示的に呼び出すこともできますが、呼び出せるのは同じモジュールの中からだけです。

■　ジェネラル プロシージャ

イベントに直接には関連付けられていないプロシージャです。イベント プロシージャでない **SUB** プロシージャと、すべての **FUNCTION** プロシージャは、ジェネラル プロシージャです。フォーム モジュール内のジェネラル プロシージャは、そのモジュールの中からだけ呼び出すことができます。一方、コード モジュール中のプロシージャはすべてジェネラル プロシージャであり、プロジェクト中のどこからでも呼び出すことができます。

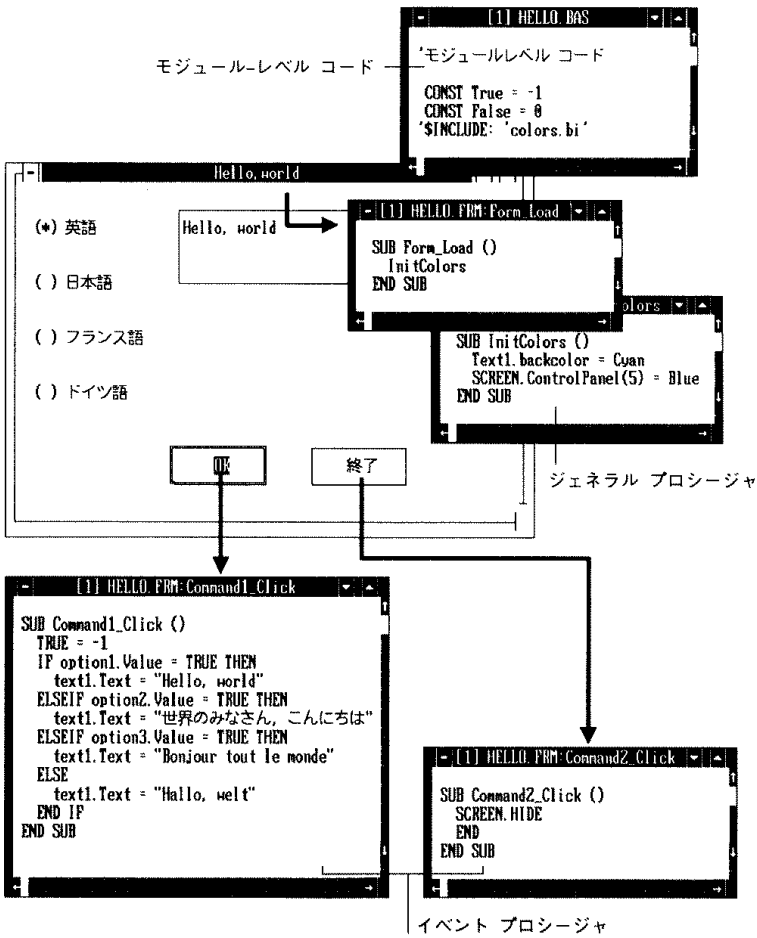


図 3.1　モジュールレベル コード、イベント プロシージャ、
ジェネラル プロシージャを含むアプリケーション

3.1 イベントドリブン アプリケーションの動作

“イベント”は、ユーザーがマウスやキーボードを操作したときや、あらかじめ設定しておいた時間が経過したとき、あるいはプログラム中からオブジェクトに対して操作を行ったときに発生します。イベントにはいくつかの種類があり、マウスのクリックやキー入力など、操作の種類によって発生するイベントが異なります。イベントが発生すると、オブジェクト (フォームやコントロール) がイベントの発生を認識して、対応するイベント プロシージャを自動的に呼び出します。このような仕組みを、“イベントドリブン”と呼びます。イベントドリブン アプリケーションは、それぞれのイベントに対する応答として、Basic コードを記述して作成します。

代表的なイベントには、次のようなものがあります。

イベント	操作の内容
Click	ユーザーがフォームやコントロールをクリックした
KeyPress	ユーザーがキーボードから文字を入力した
MouseMove	ユーザーがマウスを移動した

Visual Basic が用意している各オブジェクトは、オブジェクトの種類によって、どのイベントを認識し、どのイベントを認識しないかがあらかじめ決められています。例えばフォームやコマンド ボタンは、自分自身がマウスでクリックされたことをイベントとして認識します。オブジェクトとイベントの組み合わせに従い、行いたい処理をイベント プロシージャとして記述することによって、あるイベントに対する応答方法を個々に指定して行きます。

例えば、ユーザーがオブジェクトをクリックしたときに発生するイベントは、Click イベントと呼ばれます。ユーザーがフォームをクリックすると Form_Click という名前のイベント プロシージャが実行され、コマンド ボタン Command1 をクリックすると Command1_Click という名前のイベント プロシージャが実行されます。

典型的なイベントドリブン アプリケーションは、次のように動作します。

1. アプリケーションが起動されると、所定のフォームを自動的にロードして画面に表示します。アプリケーションは初期化処理を行った後、イベントの発生を待ちます。
2. イベントが発生すると、フォームまたはコントロールがそのイベントを認識します。イベントを発生させる動作には、システムによるもの (例えば、フォームがロードされた) と、ユーザーによるもの (例えば、あるキーが押された) の 2 つの場合があります。
3. 発生したイベントに対応するイベント プロシージャが定義されていれば、それが実行されます。
4. アプリケーションは、次のイベントを待ちます。

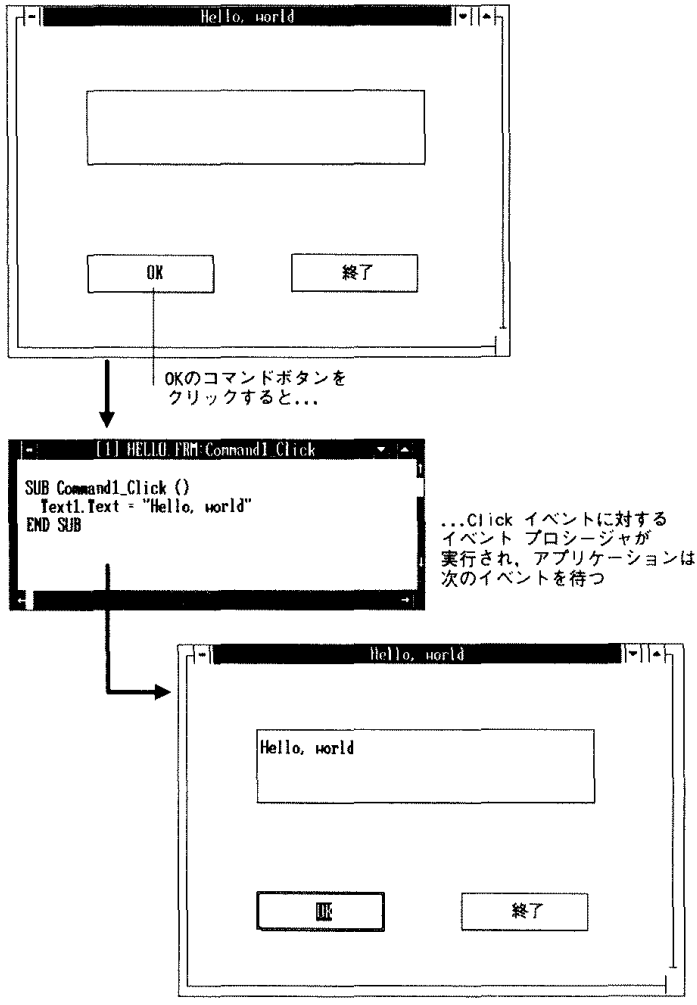


図 3.2　イベント プロシージャの実行

注　意　いくつかのイベントは、同時にほかのイベントも引き起こします。例えば、DoubleClick イベントが発生すると、MouseDown, MouseUp, Click の各イベントも同時に発生します。

3.2　イベントドリブンと従来の Basic との比較

Microsoft QuickBasic などで作成した従来の Basic アプリケーションでは、イベントではなくアプリケーション自身が、次に実行するコードの位置を決定していました。プログラムの実行は実行開始モジュールの最初の行から始まり、処理の分岐を折り込みながら、通常はソース コードに記述された順番に実行されました。

一方、Visual Basic で作成するイベントドリブン アプリケーションでは、プログラムは必要な初期化処理を行って、ユーザーインターフェイスとなるフォームを画面に表示

した後、イベントの発生を待ちます。イベントが発生すると、フォームやコントロールがそれを認識し、自動的に対応するイベント プロシージャが呼び出されます。多くの場合、どんなイベントが発生するかはユーザーの操作によって決まります。プログラムは、ユーザーが次にどんな操作を行うかによって、行う処理を選択することになります。どの処理を行う (イベント プロシージャを実行する) かの判断は、Visual Basic が自動的に行います。

従来の Basic プログラムと Visual Basic で作成するプログラムでは、プロシージャやモジュールの組み合わせ方も異なります。従来の Basic プログラムにはオブジェクトやイベントといった概念はなかったので、プロシージャはすべてジェネラル プロシージャであり、モジュールはすべてコード モジュールでした。Visual Basic で作成するイベントドリブン プログラムでは、アプリケーションはフォーム モジュールとコード モジュールによって構成されます (ただしコード モジュールはなくてもかまいません)。フォーム モジュールには、フォーム自身とフォーム上のコントロールの情報、およびそれらのオブジェクトのためのイベント プロシージャが含まれます。また、ジェネラル プロシージャも、フォーム モジュールに含めることができます。

また、従来の Basic プログラムのプロシージャと異なり、フォーム モジュール中のプロシージャはそのモジュールに対してローカルであり、他のモジュールから直接呼び出すことはできません。一方、コード モジュール中のプロシージャ (すべてジェネラル プロシージャです) は、アプリケーション内のどこからでもアクセスできます。一般的に、小規模なアプリケーションなら、1 つのフォーム モジュールだけで構成できます。Visual Basic のプロシージャについての詳細は、第 16 章「プロシージャによる構造化プログラミング」を参照してください。

従来の Basic アプリケーションをイベントドリブン アプリケーションに変更する方法については、『機種依存情報』を参照してください。

ここでは、従来の Basic アプリケーションとイベントドリブン アプリケーションの違いを、プログラム例を通して比べてみます。例として、年齢と保険の種類に応じて、保険料を計算するプログラムを考えてみます。説明を簡単にするために、エラー処理や実際の計算部分は省きます。どちらのアプリケーションも、実際の保険料の計算は **FUNCTION** プロシージャ (Calculate) を使います。

処理	従来の Basic アプリケーション	イベントドリブン アプリケーション
ユーザーが年齢と保険番号を入力する	**INPUT** ステートメントを使って変数に値を割り当てる	テキスト ボックスのテキスト プロパティを読んで変数に値を割り当てる
ユーザーの入力が終了して計算を始める	ユーザーが [ENTER] キーを押す	コマンド ボタンがクリックされると、イベント プロシージャ が呼び出される
保険の金額を表示する	**PRINT** ステートメントを使う	**MSGBOX** 関数またはステートメントを使ってメッセージ ボックスを表示する

従来の Basic アプリケーション

従来の Basic アプリケーションでは、モジュールレベル コードでユーザーに 2 つの値の入力を求めた後、Calculate プロシージャを呼び出して保険料の金額を計算し、表示します。実行時には、次のように表示されます。

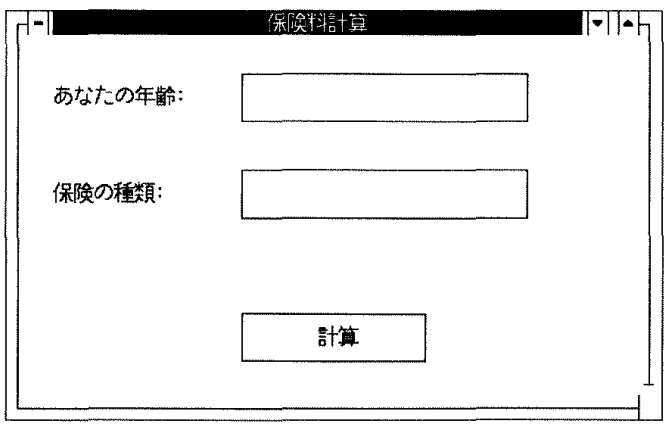
```
あなたの年令を入力してください:  40
保険の種類を入力してください:    E9275
あなたの年間の保険料は           8000
```

このアプリケーションのコードは、モジュールレベル コードと 1 つのプロシージャから構成されます。

```
'モジュールレベル コード
INPUT "あなたの年令: ", Age%
INPUT "保険の種類: ", Policy$
Amt% = Calculate% (Age%, Policy$)
PRINT "あなたの年間の保険料は", Amt%

'Calculate プロシージャ
FUNCTION Calculate% (Age AS INTEGER, Policy AS STRING)
'保険料を計算する
.
.
.
END SUB
```

イベントドリブン アプリケーション

同じ機能を持つイベントドリブン アプリケーションは、まずフォームを表示します。ユーザーは、txtGetAge と txtGetPolicy という名前の 2 つのテキスト ボックスに値を入力します。cmdCalculate という名前のコマンド ボタンをクリックすると、保険料が計算されて画面に表示されます。



このアプリケーションは、Click イベントに対するイベント プロシージャと、保険料を計算する 1 つのジェネラル プロシージャから構成されます。

```
'Click イベントに対するイベント プロシージャ
SUB cmdCalculate_Click
    Age% = VAL(txtGetAge.Text)        '年令の文字列を数値に変換する
    Policy$ = txtGetPolicy.Text       '保険の名前を得る

    Amt% = Calculate%(Age%, Policy$)
    MSGBOX "あなたの年間の保険料は " + STR$(Amt%) '保険料を表示する
END SUB

'Calculate プロシージャ
FUNCTION Calculate% (Age AS INTEGER, Policy AS STRING)
'保険料を計算する
.
.
.
END SUB
```

画面の 1 番上のテキスト ボックスに入力された年令 (txtGetAge の Text プロパティ) は、**VAL** ステートメントによって数値に変換された後、変数 Age% に代入されます。保険の金額の計算が完了すると、**STR$** 関数を使って金額 (Amt%) を文字列に変換し、メッセージ ボックスに表示します。

このアプリケーションにはモジュールレベル コードが存在しない点に注意してください。一般的なイベントドリブン アプリケーションは、最初のフォームを画面に表示するところから処理を開始します。

3.3 インターフェイスの設計

イベントドリブン アプリケーションの作成は、ユーザー インターフェイスの設計から始めます。具体的には、アプリケーションの実行時にユーザーに対するインターフェイスとなるオブジェクトを、フォーム上に配置する作業がこれに当たります。

Visual Basic のオブジェクト

Visual Basic のオブジェクトには、フォーム，いくつかのコントロール，および 3 つの特殊オブジェクトがあります。

オブジェクト	説明
フォーム	ウィンドウやダイアログ ボックスなど、コントロールを表示する枠組みとなるものです。フォームとそれに含まれるコントロールは、フォーム デザイナを使って作成します。**INPUTBOX$** 関数と **MSGBOX** 関数またはステートメントを使って、定義済みのフォームを利用することもできます。
コントロール	テキスト ボックスやコマンド ボタンなど、フォーム上に配置されるオブジェクトです。各コントロールには、設定できるプロパティと認識できるイベントのセットがあらかじめ決められています。

SCREEN オブジェクト	画面を表す特殊オブジェクトです。**SCREEN** オブジェクトの ControlPanel プロパティの設定を変更することによって、境界線、画面の背景 (デスクトップ)、スクロールバー、タイトル バーの色などの属性を設定できます。詳しくは第 14 章「アプリケーションの外観の変更」を参照してください。
CLIPBOARD オブジェクト	クリップボードを表す特殊オブジェクトです。テキストの切り抜き，複写，貼り付けを行うときに使います。詳しくは、第 7 章「テキストの処理」を参照してください。
PRINTER オブジェクト	プリンタを表す特殊オブジェクトです。

注　意　フォームとコントロールは、デザイン時にフォーム デザイナを使って作成しますが、実行時にプログラムからアクセスすることも可能です。一方、**SCREEN, CLIPBOARD, PRINTER** の各オブジェクトは、実行時にプログラムからだけアクセスできます。

フォームの作成と編集

まず、アプリケーションで使うウィンドウやダイアログ ボックスのベースとなるフォームを作成します。フォームを作成するには、プログラミング環境から [ﾌｧｲﾙ] メニューの [新規フォーム] コマンドを選びます。フォーム デザイナで 1 度に編集できるフォームは 1 つだけです。

新しく作成して保存したフォームは、自動的に現在作業中のプロジェクトに追加されます。別のプロジェクトにフォームを登録したいときは、新しいプロジェクトをオープンする必要があります。

既存のフォームを編集してサイズの変更や移動を行うには、プログラミング環境上で、[表示] メニューの [フォーム選択] コマンドを選びます。[フォーム選択] ダイアログ ボックスには、現在作業中のプロジェクトに含まれるフォームの一覧が表示されます。この中から編集するフォームを選ぶと、フォーム デザイナが起動して、そのフォームが表示されます。

▼ **新しいフォームを作成するには**

- [ﾌｧｲﾙ] メニューの [新規フォーム] コマンドを選びます。

  新しいフォームが、画面の中央に表示されます。

▼ **現在作業中のプロジェクトに含まれる既存のフォームを編集するには**

1. [表示] メニューの [フォーム選択] コマンドを選びます。

   プロジェクトに含まれるフォームの一覧が表示されます。

2. 編集するフォームを選んで <了解> をクリックします。

▼ 新しいプロジェクトを作成するには

- [ﾌｧｲﾙ] メニューの [新規プロジェクト] コマンドを選びます。

▼ 既存のプロジェクトをオープンするには

1. [ﾌｧｲﾙ] メニューの [プロジェクト オープン] コマンドを選びます。

   [プロジェクト オープン] コマンド ダイアログ ボックスが表示されます。すべてのプロジェクト ファイルは拡張子 .MAK であることに注意してください。プロジェクトが 1 つのフォーム モジュールだけで構成されている場合は、テキスト ボックスにそのフォーム モジュールの名前を直接入力してください。

2. プロジェクトの名前を入力するか、ファイル リスト ボックスからプロジェクトを選択して <了解> をクリックします。

フォームへのコントロールの配置

フォーム デザイナでは、ツールボックスか [ツール] メニューを使って生成したいコントロールを選択し、フォーム上に配置して行きます。

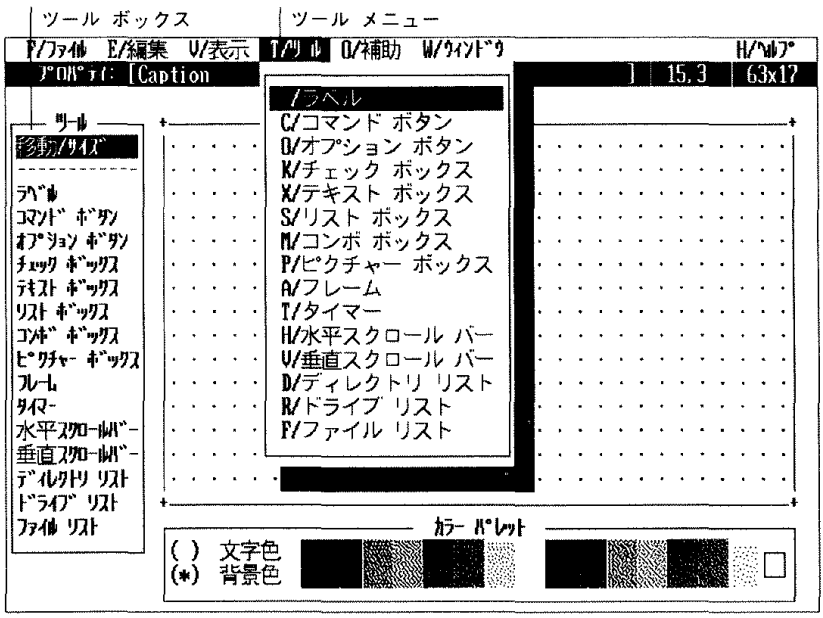


図 3.3 ツールボックスと [ツール] メニュー

参 考 コントロール配列と呼ばれる機能を使うと、実行時にフォームにコントロールを追加することができます。コントロール配列の使い方については、第 4 章「コントロールの作成と利用」を参照してください。

▼ **フォームに新しくコントロールを追加するには**

- [ツール] メニューからコントロールを選択するか、ツールボックスに一覧表示されているコントロール名をダブルクリックします。この操作で、デフォルトのサイズのコントロールがフォームの中央付近に配置されます。

  または、

  ツールボックス上のコントロールの名前をクリックし、希望の位置とサイズになるようにマウスをドラッグします。

新しくコントロールがフォーム上に配置されると、コントロールの四隅に、“サイズハンドル” と呼ばれる "+" のキャラクタが表示されます。

```
| サイズ ハンドル
+-----------+
|Text1      |
+-----------+
```

オブジェクトの選択

オブジェクトの移動，サイズ変更，削除，プロパティ設定を行うには、まず目的のオブジェクトを選択します。選択されたオブジェクトは四隅にサイズ ハンドルが表示されます。

▼ **オブジェクトを選択するには**

- オブジェクトをクリックします。

  または、

  目的のオブジェクトが選択されるまで [TAB] キーを押します。[TAB] キーを押すと、フォーム上のオブジェクトが順に選択されます。

▼ **複数のコントロールを一度に選択するには (マウスが必要です)**

1. コントロールをクリックします。
2. [CTRL] を押しながら次のコントロールをクリックします。

   両方のコントロールに、サイズ ハンドルが表示されます。
3. 同じ手順を繰り返して、同時に選択したいコントロールをすべて指定します。

この操作で選択した複数のコントロールは、同時に移動したり削除したりできます。複数のコントロールを選択しているときは、コントロールのサイズ変更やプロパティ設定はできません。

オブジェクトの移動とサイズ変更

▼ **オブジェクトを移動するには**

1. オブジェクトを選びます。
2. マウス ポインタをオブジェクトのサイズ ハンドル以外の部分に合わせ、新しい位置までドラッグします。

   または、

   方向キーを押します。[CTRL] + 方向キーを押すとより速く移動します。

▼ **オブジェクトのサイズを変更するには**

1. オブジェクトを選びます。
2. マウス ポインタをサイズ ハンドルに合わせ、希望のサイズになるまでマウスをドラッグします。

   または、

   [SHIFT] + 方向キーを押します。[SHIFT] + [CTRL]+ 方向キーを押すとより速くサイズが変わります。

▼ **フォームを最大化するには**

- フォームの最大化ボタンをクリックします。

  または、

  [CTRL] + [F10] キーを押します。

フォームを最大化すると画面全体に表示され、サイズ ハンドルで大きさを変更できなくなります。最大化したフォームを元のサイズに戻すには、フォームのコントロール メニュー ボックスから [復元] コマンド を選ぶか、[CTRL] + [F5] キーを押します。

参 考 フォームのサイズを変更したり最大化したときに、メニュー バー，ツールボックス，カラー パレットが隠れて見えなくなることがあります。このような場合は、次のような操作を行ってください。

- メニュー バーとプロパティ バーを表示したいときは、[ﾒﾆｭｰ] キーを押します。
- フォームにコントロールを追加するときは、ツールボックスの代わりに [ﾂｰﾙ] メニューを使います。
- カラー パレットを表示するには、[ｳｨﾝﾄﾞｳ] メニューの [カラー パレット] コマンドを選びます。

コントロールとフォームの削除

▼ フォームからコントロールを削除するには

1. フォーム デザイナ上で、削除するコントロールを選びます。
2. [編集] メニューから [消去] を選ぶか、[DEL] キーを押します。

▼ プロジェクトからフォーム モジュールを解放するには

1. [ﾌｧｲﾙ] メニューの [フォーム解放] コマンドを選びます。

   [フォーム解放] ダイアログ ボックスが表示されます。
2. 削除するフォームの名前を選びます。
3. <了解> を選ぶと、そのフォーム モジュールはプロジェクトから除かれます。ただし、ファイル自体が削除されるわけではありません。

3.4　プロパティの設定

すべてのオブジェクトは、“プロパティ”と呼ばれる属性を持っています。各プロパティにはデフォルト値が決められていますが、アプリケーションに合わせて値を変更することができます。

オブジェクトの外観と動作は、これらのプロパティによって決まります。オブジェクトの名前，内容，サイズ，位置，色などの特性も、プロパティの一部です。例えば、コマンド ボタンは次に示すプロパティを持っています。

コマンドボタンのプロパティ

BackColor	Cancel	Caption	CtlName	Default
DragMode	Enabled	Height	Index	Left
MousePointer	TabIndex	TabStop	Tag	Top
Visible	Width			

オブジェクトの種類によって、持っているプロパティも異なります。例えば、ほとんどのオブジェクトは Caption プロパティを持っていますが、FormName プロパティを持っているのはフォームだけです。オブジェクトとプロパティの関係の一覧は、『リファレンス』またはオンライン ヘルプを参照してください。

プロパティは、デザイン時にフォーム デザイナを使って設定することもできれば、実行段階でプログラムから設定することもできます。ただし、一部のプロパティは実行時に変更できないなどの制限があります。

代表的なプロパティには、次のようなものがあります。

プロパティ	説明
CtlName	コード中で参照するためのコントロールの名前を示します。
FormName	コード中で参照するためのフォームの名前を示します。
BackColor	オブジェクトの背景色を示します。
Caption	コントロールの内側やフォームのタイトル バーに表示されるテキスト文字列を示します。デフォルトでは FormName または CtlName と同じ設定になっていますが、変更することもできます。
Text	テキスト ボックスに表示されるテキスト文字列を示します。

デフォルトの値は、多くの場合、そのまま利用できるように設定されていますが、必要に応じてデフォルトの値を変更し、アプリケーションのユーザーインターフェイスを改良してください。例えば、ラベル コントロールはデフォルトでは境界線が表示されません (BorderStyle = 0)。境界線を表示したいときは、BorderStyle プロパティを 1 または 2 に設定してください。

フォーム デザイナにおけるプロパティの設定

デザイン時にプロパティを設定するには、フォーム デザイナのプロパティ バーを使います。通常、プロパティ バーは画面にいつも表示されていますが、フォームを最大化していると、その下に隠れています。

▼ **プロパティ バーを画面に表示するには**

- プロパティ バーを一時的に表示するには、[メニュー] キーを押します。

  または、

  [値] ボックスに入力可能な状態にするには、[F2] キーを押します。[TAB] キーを押すと、[プロパティ] リスト ボックスと [値] ボックスの間を行き来できます。

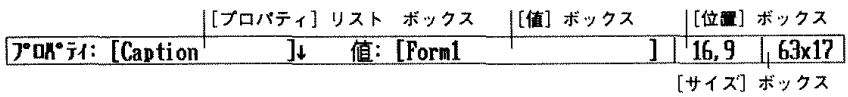


図 3.4 プロパティ バー

プロパティ バーは、次の 3 つの部分から構成されます。

部分	機能
[プロパティ] リスト ボックス	デザイン時に、選択したオブジェクトに対して設定可能なプロパティの一覧を表示します。右側にある矢印をクリックすると、リスト ボックスがドロップダウンします。また、[↓] キーを押すとプロパティが順に表示されます。
[値] ボックス	現在のプロパティ設定を表示したり、新しい設定を入力するための領域です。True/False など、そのプロパティに対して設定できる値が 決まっているときは、右側に矢印が表示され、リスト ボックスになります。このときは、ボックスに直接入力することはできません。名前などを入力するときは、矢印が表示されずにテキスト ボックスになります。値を直接入力してください。
位置およびサイズ ボックス	選択されているオブジェクトの位置とサイズを示します 。

▼ プロパティの設定を変更するには

1. オブジェクトを選びます。
2. [プロパティ] リスト ボックスからプロパティを選びます。リスト内のスクロールには方向キーまたは垂直スクロール バーを使います。
3. [値] ボックスに設定値を入力するか、ドロップダウンするリストから値を選んで [ENTER] キーを押します。また、[↑] および [↓] キーで選択することもできます。

   新しく設定したプロパティの値に従って、オブジェクトが変更されます。

ヒント　プロパティをすばやく設定するには、プロパティ リスト ボックスの右側の矢印をクリックしてリストをドロップダウンして、設定したいプロパティの先頭の文字 (例えば Width であれば W) のキーを押してください。その文字で始まる最初のプロパティが選択されます。もう 1 度同じキーを押すと、その文字で始まる次のプロパティに移ります。もちろん、ここでも方向キーは有効です。

注　意　FormName および CtlName プロパティに設定されている文字列は、コード中でフォームおよびコントロールの名前として機能します。これらのプロパティを変更するとき、文字列にスペースが含まれないように注意してください。文字列がスペースを含む場合、そのフォームまたはコントロールに対するコードが記述できなくなります。

プログラム中からのプロパティの設定

オブジェクトのプロパティを変更する方法は、デザイン時にフォーム デザイナのプロパティ バーを使う方法だけではありません。実行時にプログラム中から設定したり、ユーザーからの入力に従って変更したりすることも可能です。

コード中では、プロパティは次の構文を使って示します。

object.property

いくつかのプロパティに対しては、ユーザーが入力した文字列を実行時に代入することもできます。例えば、コマンド ボタンに表示されるキャプションは、そのボタンの Caption プロパティに文字列を代入することによって設定できます。

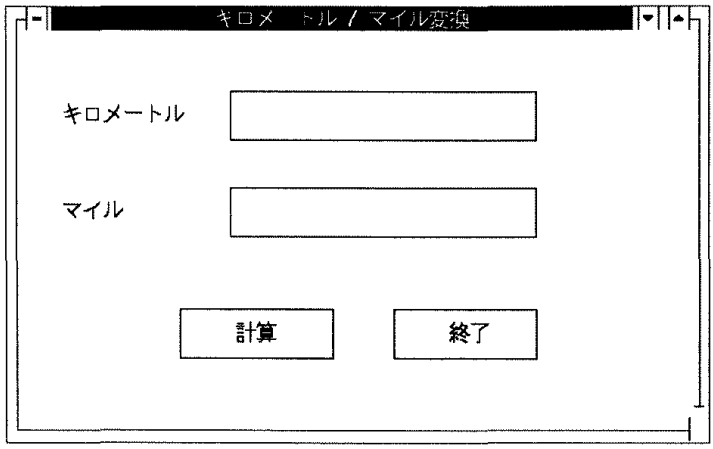
```
Command1.Caption = "押してください"
```

プロパティの設定方法の比較

テキスト ボックスの Text プロパティに値を設定する場合を考えながら、プロパティの設定方法を比較して見ることにします。次の表は、Text プロパティを変更する方法です。

行いたい操作	テキスト プロパティの設定方法
テキスト ボックスに最初に表示する文字列を決める。	デザイン時に、プロパティ バーを使って文字列を設定する。
ユーザーが入力した内容を表示する。	ユーザーの入力によって、自動的に Text プロパティが再設定されるようにしておく。
特定のイベントに対応して (例えばコマンド ボタンがクリックされたら) テキストを表示する。	プログラム中のある場所が実行されたときに、Text プロパティに値を代入するようなコードを書く。

例えば次に示すアプリケーションでは、ユーザーはキロメートルとマイルのどちらのテキスト ボックスに値を入力してもかまいません。<計算> ボタンが押されたら、どちらのテキスト ボックスに値が入力されているかを判断して、その値をもとにもう一方の値を計算し、テキストボックスに表示します。



このアプリケーションは、具体的には次のような操作を行っています。

1. 2 つのテキスト ボックスの Text プロパティを、**VAL** 関数を使って数値に変換し、Miles と Kilos の 2 つの変数にそれぞれ代入します。
2. txtKilos テキスト ボックスに、計算のもとなる値が含まれているかどうかを調べます。含まれていなければ、txtMiles テキスト ボックスに入力されたマイル数からキロメートル数を計算し、結果を txtKilos テキスト ボックスの Text プロパティに代入します。
3. txtKilos テキスト ボックスに値が入力されていれば、キロメートル数からマイル数を計算して、結果を txtMiles テキスト ボックスの Text プロパティに代入します。

これらの 2 つのテキスト ボックスの Text プロパティは、次の 3 つの方法で設定されます。

- それぞれのテキスト ボックスの Text プロパティは、フォーム デザイナのプロパティ バーを使って null 文字列 ("") に設定されています。実行開始時にはテキスト ボックスに何も表示されません。
- アプリケーションの実行中に、ユーザーがキロメートルとマイルのどちらかのテキスト ボックスに値を入力すると、対応する Text プロパティには入力した値が設定されます。
- ユーザーが <計算> ボタンをクリックすると、もう一方のテキスト ボックスの Text プロパティの値が計算され、代入文によってプロパティに設定されます。

このアプリケーションにおける、<計算> ボタン (Calculate) の Click イベント プロシージャの内容は、次のようになります。

```
SUB Calculate_Click ()
    Miles = VAL(txtMiles.text)  'テキストボックスの値を数値に変換する
    Kilos = VAL(txtKilos.text)
    IF Kilos = 0 THEN
        Kilos = Miles * 1.61            'キロメートル数を計算する
        '結果をテキスト ボックスに表示する
        txtKilos.Text = FORMAT$(Kilos, "##.00")
```

```
    ELSE
        Miles = Kilos * 0.62            'マイル数を計算する
        '結果をテキスト ボックスに表示する
        txtMiles.Text = FORMAT$(Miles, "##.00")
    END IF
END SUB
```

FORMAT$ 関数は、Text プロパティに代入する前に、数値を再び文字列に変換し、文字列式で指定された書式にします。

サイズと位置を決めるプロパティの設定

フォームやコントロールのサイズと位置を決めるプロパティ (Left, Top, Height, Width) は、対象となるフォームやコントロールを移動したりサイズを変更したりすると自動的に変更されます。また、デザイン時にプロパティ バーを使って設定することもできます。

選択されているオブジェクトの位置とサイズは、プロパティ バーに表示されます。左のボックスにはオブジェクトの左上角の座標、右のボックスにはオブジェクトのサイズが、それぞれ表示されます (図 3.4 を参照してください)。

これらのボックスに値を直接入力することはできませんが、他のプロパティと同じ方法で位置とサイズのプロパティを変更できます。例えばテキスト ボックスの高さを変更したいときは、対象のテキスト ボックスを選んで Height プロパティを選択し、[値] ボックスに新しい値を入力します。

3.5 Basic コードの記述

アプリケーションのユーザー インターフェイスの設計が完了したら、次にプログラミング環境に移って実際にコードを書いて行きます。ここでは、Visual Basic のプログラムに登場する典型的なコードについて説明します。

プロシージャを使った操作

Visual Basic のプログラミングでは、QuickBasic や Microsoft Basic と同様に、コードを“プロシージャ”と呼ばれる単位に分けて管理します。プロシージャの分類には、2 種類の方法があります。

値を返すかどうかという点から分類すると、**SUB** プロシージャと **FUNCTION** プロシージャに分けることができます。

- **SUB** プロシージャは、値を返さないプロシージャです。**CALL** ステートメントで呼び出すか、宣言した後にプロシージャの名前を直接指定して呼び出します。
- **FUNCTION** プロシージャは、値を返すプロシージャです。式の中にそのプロシージャの名前を含めることによって呼び出します。

また、別の分類方法では、イベント プロシージャとジェネラル プロシージャに分けられます。

- イベント プロシージャは、イベントが発生したときに自動的に呼び出されます。必ずフォーム モジュールに含まれます。また、値を返さない **SUB** プロシージャの一種です。**CALL** ステートメントを使って、明示的にイベント プロシージャを呼び出すこともできます。
- イベント プロシージャ以外のすべてのプロシージャは、ジェネラル プロシージャと呼ばれます。フォーム モジュールとコード モジュールのどちらに含めることもできますし、**SUB** または **FUNCTION** プロシージャのどちらであってもかまいません。コード モジュール中のすべてのモジュールはジェネラル プロシージャです。また、**FUNCTION** プロシージャは常にジェネラル プロシージャです。ジェネラル プロシージャは、プログラム中から明示的に呼び出されたときだけ実行されます。

フォーム モジュール内では、すべてのプロシージャは (ジェネラル プロシージャもイベント プロシージャも)、そのモジュールに対してローカルです。つまり、そのフォーム モジュール内からしか呼び出すことができません。例えば、フォーム モジュール FORM1.FRM に含まれる Command1_Click プロシージャは、別のフォーム モジュール FORM2.FRM から直接呼び出すことはできません (ただし、FORM2.FRM から操作を行うことによってイベントを発生させ、間接的に実行させることはできます)。また FORM2.FRM に Command1_Click プロシージャがある場合には、FORM2.FRM に含まれる Command1_Click プロシージャとは別のプロシージャとして扱われます。

一方、コード モジュール内のプロシージャ (常にジェネラル プロシージャです) は、アプリケーション内のどこからでも呼び出すことができます。このため、ジェネラル プロシージャに対しては、同じアプリケーション内で重複しないように名前を付ける必要があります。まったく同じコードが複数のモジュールに含まれる場合は、1 つのジェネラル プロシージャにまとめることをおすすめします。

プログラミング環境では、次の表に示すように、プロシージャに対する各種の操作を実行できます。

目的	メニュー コマンド/テクニック
ジェネラル プロシージャを作成する	[編集] メニューの [新規 SUB プロシージャ] または [新規 FUNCTION プロシージャ] コマンドを選ぶ。 または、 コード ウィンドウで、**SUB** *name* または **FUNCTION** *name* と入力する。
イベント プロシージャを作成または編集する	[編集] メニューの [イベント プロシージャ] コマンドを選ぶ。
プロジェクトに含まれるプロシージャの一覧を表示する	[表示] メニューの [コード選択] コマンドを選ぶ。
プロシージャを編集する	[表示] メニューの [コード選択] コマンドを選び、編集したいプロシージャの名前を選択して、<編集> または <新規ウィンドウ編集> ボタンをクリックする。

プロジェクトからプロシージャを除く	[表示] メニューの [コード選択] コマンドを選び、削除したいプロシージャの名前を選択して <削除> ボタンをクリックする。
モジュール間でプロシージャを移動する	[表示] メニューの [コード選択] コマンドを選び、移動したいプロシージャの名前を選択して <移動> ボタンをクリックする。
フォーム モジュール間でプロシージャを複写する	コード ウィンドウ内でプロシージャを選択し、[編集] メニューの [複写] コマンドを選ぶ。次に、コピー先のモジュールをアクティブにし、[編集] メニューの [貼り付け] コマンドを選ぶ。

プロシージャの詳細については、第 16 章「プロシージャによる構造化プログラミング」を参照してください。

イベント プロシージャの作成

あるオブジェクトに対するイベント プロシージャを作成するには、[編集] メニューの [イベント プロシージャ] コマンドを選びます。[イベント プロシージャ] ダイアログ ボックスには、次の 3 つのリスト ボックスが表示されます。

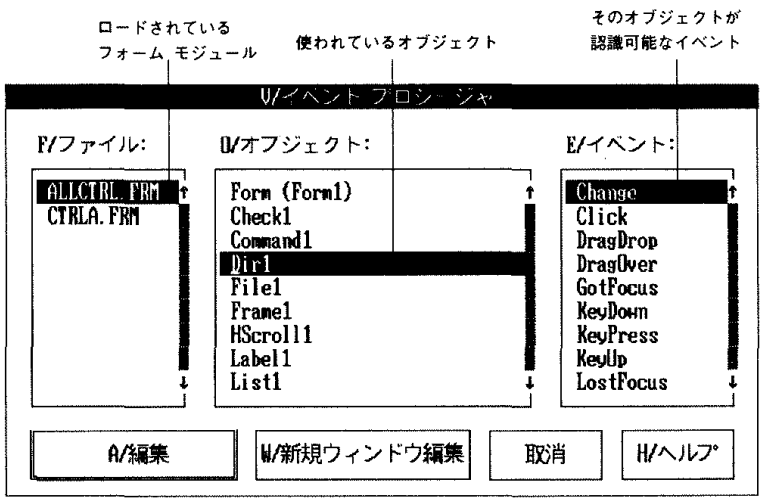


図 3.5 [イベント プロシージャ] ダイアログ ボックス

これらのリストから項目を選択して新しくイベント プロシージャを作成すると、コード ウィンドウにイベント プロシージャのテンプレートが表示されます。

ヒント コード ウィンドウでプロシージャ名を直接入力してもイベント プロシージャの作成を開始できますが、Visual Basic が用意するテンプレートを利用した方が、自動的に正しいプロシージャ名の設定と引数の宣言が行われるので便利です。

ここでは、フォーム モジュール NEW.FRM 中のコマンド ボタン cmdNew に対応するイベント プロシージャを記述する例を示します。

▼ イベント プロシージャを作成するには

1. [編集] メニューの [イベント プロシージャ] コマンドを選びます。

   [イベント プロシージャ] ダイアログ ボックスが表示されます。

2. 作成するイベント プロシージャの対象となるフォーム，オブジェクト，イベントを、それぞれリスト ボックスから選びます。

   ヒント　すでに対応するイベント プロシージャが作成されているイベントの名前は、[イベント] リスト ボックスに大文字で表示されます。

3. <編集> ボタンをクリックします。アクティブなコード ウィンドウにテンプレートが表示されます。また、<新規ウィンドウ編集> ボタンをクリックすると、新しいコード ウィンドウにテンプレートが表示されます。

重　要　イベント プロシージャを記述した後で、フォーム デザイナ上でコントロールの名前 (CtlName プロパティ) を変更した場合は、新しいコントロール名と一致するようにプロシージャ名も変更する必要があります。

イベント プロシージャの構文

イベント プロシージャの一般的な構文は、次のとおりです。

コントロールに関するイベント プロシージャ	フォームに関するイベント プロシージャ
SUB *CtlName_eventname* () *statementblock* **END SUB**	**SUB Form**_*eventname* () *statementblock* **END SUB**

SUB と **END SUB** は、それぞれイベント プロシージャの先頭と最後を表します。プロシージャの名前は、そのプロシージャの対象がコントロールのときはコントロール名 (*CtlName* プロパティに設定されている名前) で始まり、フォームのときは **Form** で始まります。その直後にアンダースコア (_)、その後にイベント名 (*eventname*) が続きます。例えば、第 2 章の Hello, world アプリケーションでは、コマンド ボタンの Click イベントに対するイベント プロシージャを記述しましたが、その場合はコマンド ボタンの名前 (CtrlName) が Command1 でイベントが Click であるため、イベント プロシージャの名前は Command1_Click になります。

statementblock には、イベントが発生したときに実行したいコードを記述します。

パラメータ リストを持つイベント プロシージャ

イベント プロシージャの中には、パラメータ (仮引数) の指定が必要となるものがあります。例えば、フォームがフォーカスを持っているときにユーザーがキーを押すと Form_KeyPress イベント プロシージャが起動されますが、このプロシージャの構文は次のようになっています。

```
SUB Form_KeyPress(KeyCode AS INTEGER)
```

この構文で、*KeyCode* は、押されたキーを引き渡すための変数です。Form_KeyPress イベントが発生すると、このプロシージャが呼び出され、押されたキーの ASCII コードが *KeyCode* に指定した変数に代入されます。

注 意 イベント プロシージャに渡されるパラメータの個数とデータ型は変更できません。イベント プロシージャに対してさらに別の情報を渡すには、Tag プロパティの設定など別の手段を利用する必要があります。

モジュールレベル コードに対する宣言と定義の追加

フォーム モジュールのモジュールレベル コードに、実行可能なコード (実行可能ステートメント) を含めることはできません。ここには、そのモジュールに対してグローバルな定数と変数の宣言だけを書きます。

```
CONST TRUE = -1
CONST FALSE = 0
DIM SHARED SaveState AS INTEGER
```

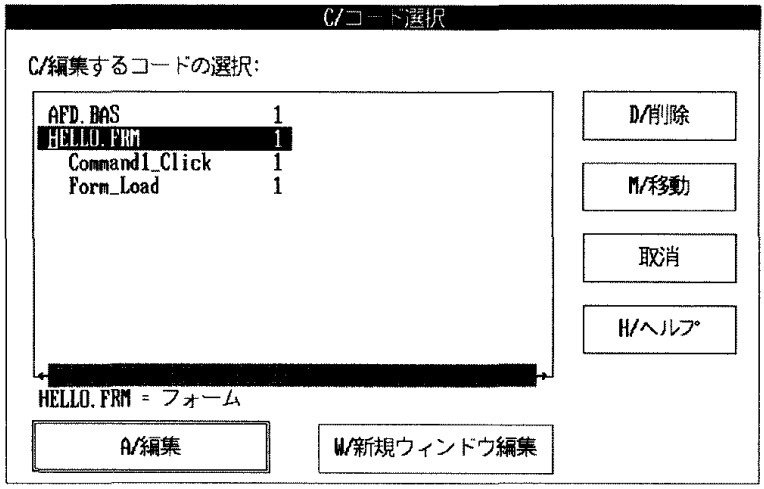
▼ **フォームのモジュールレベル コードを表示するには**

1. [表示] メニューの [コード選択] コマンドを選びます。

   [コード選択] ダイアログ ボックスが表示されます。

2. フォーム モジュールのファイル名を選びます (プロシージャの名前ではありません)。

例えば、次のダイアログ ボックスで HELLO.FRM を選ぶと、HELLO.FRM のモジュールレベル コードが表示されます。



注　意　コード モジュールのモジュールレベルに実行可能なコードを記述することは可能です。ただしそのモジュールが実行開始モジュールでない限り、通常はそれらのコードは実行されません。

Form_Load プロシージャへのコードの追加

Form_Load イベントは、フォームがメモリにロードされるたびに発生します。Form_Load イベント プロシージャには、通常はフォームの初期化コードを記述します。例えば、デフォルト カラーなどの初期値の設定やリスト ボックスへの項目の追加などの処理を行います。

Form_Load イベントは、実際には次に示す 5 つのイベントを生成します。

- Form_Load
- Form_Resize
- *object*_GotFocus (フォーム上にコントロールが配置されておらず、オブジェクトがフォーカスを持つことができる場合)
- Form_Paint
- *picture*_Paint (フォームがピクチャー ボックスを含んでいる場合)

参　考　デフォルトでは、Form_Load プロシージャが実行された直後にフォーカスを持っているのは、TabIndex プロパティが 1 に設定されているコントロールです。それ以外のコントロールにフォーカスを移すには、**SETFOCUS** メソッドを使います。

複数のアプリケーションで同じ初期化コードを実行したい場合は、コード モジュールにジェネラル プロシージャを 1 つ作成して、Form_Load からそれを呼び出すようにします。例えば、起動時に表示されるフォームやコントロールに、デフォルトで同じ色を割り当てたいときは、色の割り当てを行うジェネラル プロシージャ InitColors を作成し、このプロシージャを Form_Load イベント プロシージャから呼び出すようにします。

```
SUB Form_Load ( )
  .
  .      'フォームの初期化を行うコード
  .
CALL InitColors          'CALL キーワードは省略可能
END SUB
```

注 意 Form_Load プロシージャ内に **PRINT** メソッドを記述する場合は、AutoRedraw を True (-1) に設定してください。このように設定されていないと、プログラムの実行時に **PRINT** による出力が表示されません。**PRINT** メソッドを Form_Load ではなく Form_Paint イベント プロシージャ中に記述するという方法もあります。

プログラムの終了方法

イベントドリブン アプリケーションは、すべてのフォームがクローズされて実行中のコードがなくなると終了します。プログラムを終了する最も簡単な方法は、**END** ステートメントを使ってすべてのコードの実行を終了させるとともに、オープンしたすべてのファイルをクローズすることです。ただし **END** ステートメントを実行する前に、明示的にすべてのフォームをクローズすることをおすすめします。

例えば、ユーザーがコマンド ボタン cmdQuit を押したときにプログラムを終了させるには、次のような Click イベント プロシージャを作成します。

```
SUB cmdQuit_Click ( )
    SCREEN.HIDE
    END
END SUB
```

メソッドの利用

Visual Basic は、フォームやコントロールに対して直接操作を行うための方法として、“メソッド” と呼ばれる機能を用意しています。メソッドは、関数やステートメントと同様に、キーワードになっています。

メソッドの構文は、次のとおりです。

{[*FormName.*]*CtlName.*|*FormName.*|*specialobject.*}*method*

method の前に、ピリオドで区切ってそのメソッドを適用したいオブジェクト (コントロール，フォーム，または特殊オブジェクト) を指定します。オブジェクトの指定を省略すると、そのときアクティブなフォームに対して適用されます。

例えば **PRINT** メソッドを使うと、数式や文字列式の値をフォームやピクチャ ボックスに出力できます。MyForm という名前のフォームに対して文字列を出力する場合は、次のように**PRINT** メソッドを使います。

```
MyForm.PRINT "ここの文字列はフォームに出力されます"
```

プロパティと同様に、それぞれのフォームやコントロールによって、適用できるメソッドがあらかじめ決まっています。それぞれのフォームやコントロールに対してどのメソッドが適用できるかについては、『リファレンス』またはオンライン ヘルプを参照してください。

キーボード インターフェイスのサポート

多くのユーザーは、アプリケーションの操作にマウスを使います。ただし、ユーザーによってはマウスがなかったり、キーボードでの操作を好む場合もあるので、常にマウスの操作と同じ機能のキーボード操作を用意しておくことが重要です。

キーボードを使って、あるコントロールにフォーカスを移すには、次のように操作します。

- [TAB] キーを押すと、選択可能なコントロールを順にフォーカスが移動します。目的のコントロールにフォーカスが移るまで [TAB] キーを続けて押します。
- そのコントロールに対応するアクセス キーを押します。アクセス キーとは、同時に押す [ﾒﾆｭｰ] キーといずれかの文字キーの組み合わせ (例えば [ﾒﾆｭｰ] + [A]) を指します。

[SPACE] キー または [ENTER] キーを押すと、そのときフォーカスを持っているコントロールをクリックしたことと同じ意味になり、Click イベントが発生します。

さらに、メニューコマンドに対してショートカット キーを設定することができます。ショートカット キーが設定されているメニュー コマンドは、メニューをオープンしなくても、キーを押すだけで直接実行できます。通常、ショートカット キーにはファンクション キーやコントロール キーの組み合わせ (例えば [CTRL] + [F1] キー) を割り当てます。ショートカット キーについて詳しくは、第 8 章「メニューの作成」を参照してください。

アクセス キーの割り当て

アクセス キーには、なるべく意味が分かりやすい文字を割り当てるようにします。ただし、2 バイト文字やカタカナをアクセス キーにすることはできないので注意してください。例えば、[ﾌｧｲﾙ] メニューのアクセス キーは通常、F (file の先頭の文字) にします。Caption プロパティを持つすべてのコントロールと、すべてのメニュー名やメニュー コマンドに対して、アクセス キーを割り当てることができます。

▼ コントロールにアクセス キーを割り当てるには

- Caption プロパティの中で、アクセス キーに設定したい文字の前にアンパサンド (&) を付けます。

  例えばファイルに関するメニューなら、名前を "F/ﾌｧｲﾙ:" にして、先頭にアンパサンドを付けるようにします。アンパサンド自身は表示されずに、その次の文字 (この場合は F) が強調表示されます。

例えば、<O/了解> ボタンのアクセス キーが [ﾒﾆｭｰ] +[O] (OK の O)、<X/終了> ボタンのアクセス キーが [ﾒﾆｭｰ] + [X] (eXit の X) になるように、それぞれの Caption プロパティを設定した場合は、次のように表示されます。

キャプションのないコントロールへのアクセス キーの割り当て

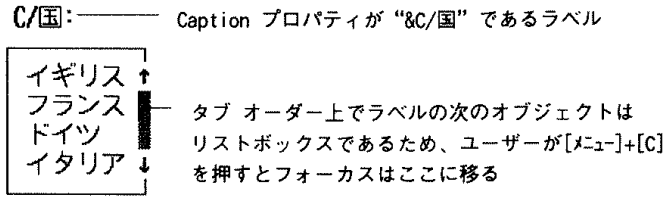
Caption プロパティを持たないコントロールに対してアクセス キーを割り当てるには、ラベル コントロールを使います。ただしラベルはフォーカスを持てないため、[TAB] キーを使ってコントロールを移動しても、フォーカスは自動的にそのコントロールを飛ばして次に移ってしまいます。テキスト ボックス，ピクチャー ボックス，リスト ボックス，コンボ ボックス，ドライブ ボックス，ディレクトリ ボックスに対してアクセス キーを割り付けるときは、次のような方法を使います。

▼ ラベルを使ってコントロールにアクセス キーを割り当てるには

1. フォーム デザイナで、最初にラベル、次にコントロールの順番にフォーム上に配置します。
   または、
   コントロールを描く順番は自由ですが、ラベルがそのコントロールの直前に来るようにタブ オーダーを設定します。タブ オーダーについては後で説明します。

2. ラベルの Caption プロパティにアンパサンド (&) を含む文字列を設定し、ラベルにアクセス キーを割り当てます。

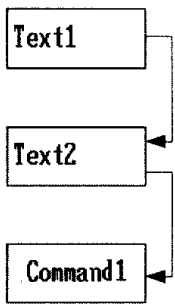
例えば、リスト ボックスに対してアクセス キーが設定されているように見せるためには、リスト ボックスの直前にアクセス キーを設定したラベルを置きます。そのアクセス キーが押されると、フォーカスはラベルに移ろうとしますが、ラベルはフォーカスを持つことができないためにスキップされ、その結果、次のリスト ボックスにフォーカスが移ります。



タブ オーダーの設定

"タブ オーダー" とは、ユーザーが [TAB] キーを押したときに、オブジェクトに対するフォーカスが移動する順番のことです。フォーカスを持つコントロールまたはフォームは強調表示され、ユーザーのキー操作はそのコントロールまたはフォームに対して意味を持ちます。例えばユーザーが [ENTER] キーを押したとき、コマンド ボタンがフォーカスを持っていれば、そのコマンド ボタンに対して Click イベントが発生します。通常は、コントロールを作成した順番がそのままタブ オーダーになります。

例として、はじめに Text1 と Text2 の 2 つのテキスト ボックスを作り、次にコマンド ボタン Command1 を作った場合を考えます。この場合、アプリケーションを起動したときは Text1 がフォーカスを持っていますが、[TAB] キーを押すたびに Text2、Command1 の順番でフォーカスが移動して行きます。



▼ **コントロールのタブ オーダーを変更するには**

- 各コントロールの TabIndex プロパティを設定します。

デフォルトでは、フォーム上に最初に配置されたコントロールの TabIndex が 0、次に配置されたコントロールが 1 というように、順番に番号が設定されています。挿入や削除によってコントロールのタブ オーダーが変化すると、新しい順番に従って自動的に番号が振り直されます。

例えば次のステートメントを実行すると、コマンド ボタン Command1 をタブ オーダーの先頭に設定できます。

```
Command1.TabIndex = 0
```

この場合、他のコントロールの TabIndex プロパティは、次の表に示すように自動的に調整されます。

コントロール	Command1 がタブ オーダーの先頭になる前の TabIndex プロパティ	Command1 をタブ オーダーの先頭に設定した後の TabIndex プロパティ
Text1	0	1
Text2	1	2
Command1	2	0

TabIndex プロパティの最大値は、“タブ オーダーの対象となるコントロールの個数 - 1” です。これは TabIndex が 0 から始まるためで、仮に TabIndex にコントロールの個数よりも大きな値を設定しても、自動的にコントロールの個数よりも 1 だけ小さい値に変換されます。また、TabIndex に負の値を設定するとエラーになります。

注 意 メニュー，タイマー，フレーム，ラベルの各コントロールは、タブ オーダーに含まれません。また、無効または非表示になっているコントロールは、フォーカスを持てません。したがって、ユーザーが [TAB] キーを押しても、これらのコントロールはスキップされます。

コントロールの Tab Stop の設定

実行時に [TAB] キーを押すと、通常はタブ オーダーに従ってフォーカスが移動します。ただし TabStop プロパティを False (0) に設定すると、そのコントロールはスキップされます。

TabStop プロパティを False に設定したプロパティは、[TAB] キーによるコントロールの選択のサイクルからスキップされるだけで、タブ オーダー上の順序は保持しています。

注 意 通常、オプション ボタンはグループ化して使いますが、グループ全体で 1 つのタブ ストップしか持ちません。選択されているボタン (つまり Value が True に設定されているボタン) の TabStop プロパティは自動的に True になり、それ以外のオプション ボタンの TabStop プロパティは False になります。

3.6 オブジェクトのフォーカスの取得と設定

“フォーカスを持っている” とは、マウスやキーボードを使って行われるユーザーからの入力を受け取ることができる状態を指します。そのときフォーカスを持っているオブジェクトは、ユーザーからの入力を受け取ることができます。例えば、あるテキスト ボックスがフォーカスを持っているとき、ユーザーがキーボードから入力した文字は、そのテキストボックスに表示されます。

オブジェクトがフォーカスを得たときは GotFocus イベント、失ったときは LostFocus イベントが発生します。

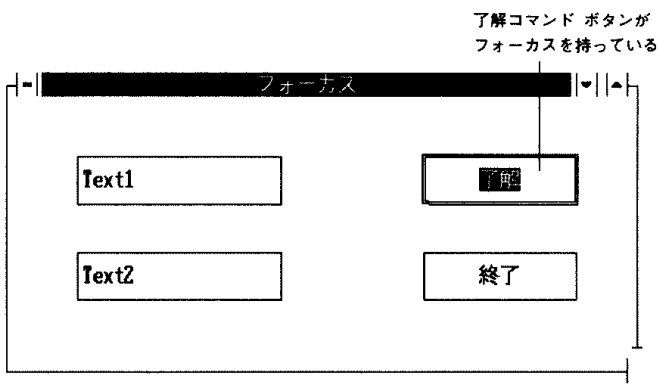
イベント	説明
GotFocus	オブジェクトがフォーカスを得たときに発生します。色を変えて表示したり、そのコントロールの操作対象を示したりするときに使います。
LostFocus	オブジェクトがフォーカスを失ったときに発生します。更新処理の確認をしたり、GotFocus に対するイベント プロシージャで変更したオブジェクトの状態を元に戻すのに使われます。

オブジェクトがフォーカスを得るには、次のようなケースがあります。

- 実行時にそのオブジェクトが選択された
- 実行時に対応するアクセス キーが押された
- プログラム中で **SETFOCUS** メソッドが実行された
- [CTRL] + [F6] キーで別のフォームにフォーカスが切り換えられた

アプリケーションが起動された直後には、起動時に表示されるフォームがアクティブであり、そのフォームで TabIndex プロパティが 0 に設定されているコントロールがフォーカスを持っています。また、アクティブなフォームに有効なコントロールがないときは、フォーム自身がフォーカスを持ちます。

そのときフォーカスを持っているオブジェクトは、画面上でも確認できます。例えば、コマンド ボタンがフォーカスを持っているときは、その境界線とキャプションが強調表示されています。



オブジェクトがフォーカスを得ることができるのは、Enabled と Visible の 2 つのプロパティがともに True (-1) に設定されているときだけです。Enabled プロパティは、そのオブジェクトがユーザーの発生させたイベント (例えばキーボードからの入力やマウスの操作) に応答できるかどうかを示しています。また Visible プロパティは、そのオブジェクトが画面上に表示されているかどうかを示しています。

注 意 フレーム，ラベル，メニュー，タイマーの各コントロールは、原則としてフォーカスを持てません。ただしフォームは、フォーカスを持つことができるコントロールを一切含んでいない場合に限って、フォーカスを持つことができます。

どの種類のコントロールがフォーカスを持っているかを知るには、**SCREEN** オブジェクトの ActiveControl プロパティを使います。例えば、フォーカスを持っているのがテキスト ボックスであるかどうかを判断するには、次に示すように ActiveControl プロパティと **IF...TYPEOF** ステートメントを使います。

```
IF TYPEOF SCREEN.ActiveControl IS TextBox THEN
    Text1.Text = "テキスト ボックスがフォーカスを持っています"
ENDIF
```

フォーム上に同じ種類のコントロールが複数ある場合は、あらかじめ各コントロールの Tag プロパティに別の文字列を設定しておき、その内容を調べてコントロールを特定するようにします。

4 コントロールの作成と利用

“コントロール” とは、ユーザーがアプリケーションを操作するために、フォーム上に配置されたオブジェクトのことを指します。Visual Basic には、15 種類のコントロールが用意されています。

これらのコントロールは、アプリケーションの中でそれぞれ次のような機能を持ちます。

コントロール	機能
チェック ボックス (Check box)	オン (チェック) / オフ (未チェック) の設定を行います。同時に複数のチェック ボックスをオンにすることができます。
コンボ ボックス (Combo box)	テキスト ボックスとリスト ボックスを組み合わせたものです。ユーザーは、ドロップダウンするリストから項目を選べます。また、キー入力によって直接設定することもできます。
コマンド ボタン (Command button)	ユーザーが実行したい命令を選択できるようにするボタンです。
ディレクトリ リスト ボックス (Directory list box)	ディレクトリ パスを表示し、ユーザーが選択できるようにします。
ドライブ リスト ボックス (Drive list box)	そのとき有効なディスク ドライブを表示し、ユーザーが選択できるようにします。
ファイル リスト ボックス (File list box)	ディレクトリ中のファイル名を表示し、ユーザーが選択できるようにします。
フレーム (Frame)	コントロールをグループ分けします。主にオプション ボタンのグループ分けに使います。
ラベル (Label)	ユーザーが直接入力や変更のできないテキストを表示します。
リスト ボックス (List box)	ドロップダウンするリストから項目を選択できるようにします。
メニュー (Menu)	メニュー コマンドを表示します。
オプション ボタン (Option button)	複数の項目から 1 つだけを選択できるようなボタンです。通常、いくつかのオプション ボタンを 1 つのグループとして使います。
ピクチャー ボックス (Picture box)	複数行におよぶテキストが表示できる領域です。ほかのコントロールをピクチャー ボックスの中に配置することができます。

水平および垂直スクロールバー (Horizontal and vertical scroll bars)	項目数の多いリストや大量の情報を表示する場合に、現在の位置を示したり設定したりするために使います。また、ある範囲で連続的に変化する値を表示したり設定したりするときにも利用できます。
テキスト ボックス (Text box)	テキストの入力や表示を行います。
タイマー (Timer)	一定の時間ごとにイベントが発生するようにします。

この章では、Visual Basic で作成するアプリケーション中でコントロールを利用する方法について説明します。『リファレンス』とオンライン ヘルプにコントロールの一覧があるので、必要に応じて参照してください。各コントロールがサポートしているメソッド, プロパティ, イベントについてもまとめてあるので、どのコントロールを使うかを決めるときの参考にしてください。

また、特定のコントロールについては、このマニュアルの次に示す各章が参考になります。

- ファイル, ドライブ, ディレクトリの各リスト ボックスについては、第 10 章「ファイルシステム コントロールの利用」を参照してください。
- メニューについては、第 8 章「メニューの作成」を参照してください。

ヒント　新しく作成したコントロールの見栄えを変える (例えば背景色や文字色の変更など) には、各コントロールのプロパティの設定を変更する他に、**SCREEN** オブジェクトのプロパティである ControlPanel 配列の設定を変更する方法もあります。詳しくは、第 14 章「アプリケーションの外観の変更」を参照してください。

4.1　コントロールの名前の付け方

新しいコントロールを作成したときは、CtlName プロパティにデフォルトのコントロール名が設定されています。例えばフォーム上に 3 つのコマンド ボタンを配置すると、それらの名前 (CtlName プロパティの設定値) は配置した順に Command1, Command2, Command3となります。

これらのデフォルトの名前をそのまま使ってもかまいませんが、似たようなコントロールの個数が増えてきたら、もっと区別しやすい別の名前をCtlName プロパティに設定したほうが便利です。

Visual Basic のマニュアルやオンラインヘルプに含まれているプログラム例では、表 4.1 に示すように、コントロールの種類を示すプリフィックスと内容を表すような名前を組み合わせて、コントロール名として使っています。このような名前にすれば、プログラムの内容がより直感的に分かりやすくなります。

表 4.1 オブジェクトの名前の付け方

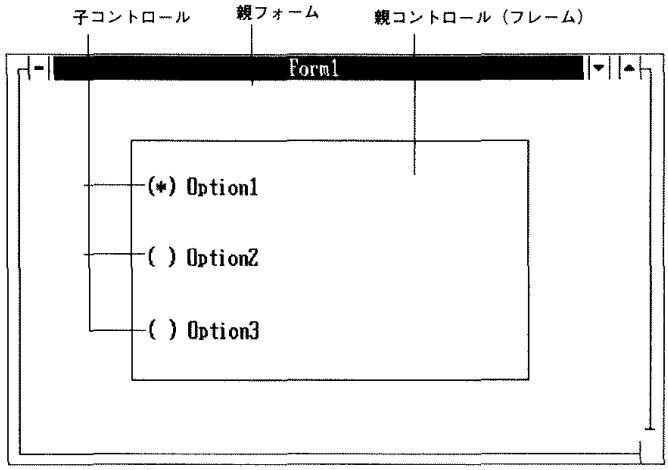
オブジェクト	プリフィックス	例
フォーム	frm	frmFileOpen
チェック ボックス	chk	chkReadOnly
コンボ ボックス	cbo	cboEnglish
コマンド ボタン	cmd	cmdCancel
ディレクトリ ボックス	dir	dirSource
ドライブ リスト ボックス	drv	drvTarget
ファイル リスト ボックス	fil	filSource
フレーム	fra	fraLanguage
ラベル	lbl	lblHelpMessage
リスト ボックス	lst	lstPolicyCodes
メニュー	mnu	mnuFileOpen
オプション ボタン	opt	optFrench
ピクチャー ボックス	pic	picDiskSpace
テキスト ボックス	txt	txtGetText
タイマー	tim	timAlarm
水平スクロール バー	hsb	hsbVolume
垂直スクロール バー	vsb	vsbRate

4.2 コントロールの親子関係

コントロールはそれぞれ独立したオブジェクトですが、フォームやコントロールの間にはある種の親子関係が存在します。

すべてのコントロールは、それが配置されたフォームに対する子の関係になり、その位置は親であるフォームに対して相対的に定義されます。具体的には、子コントロールの Left (左端) および Top (上端) プロパティが、親フォームに対する相対値で定義されます。子コントロールは親フォームの境界の外側に出ることはできませんが、右端と下端については、画面に対する絶対座標と親フォームの左上角に対する相対座標がいずれも 0～254 の範囲である限りは、親フォームの外に置けます。また、親フォームを移動すると、子コントロールもそれに合わせて移動します。

ピクチャー ボックスとフレームも、他のコントロールの親になることができます。例えば、フレーム内に配置されたオプション ボタンは、そのフレームの子になります。



子コントロールの Left と Top の 2 つのプロパティは、親コントロールに対する相対値で定義されます。子コントロールは親に合わせて移動するため、親コントロールを移動しても、子コントロールの Top と Left の各プロパティは変化しません。

フォーム デザイナにおいてピクチャー ボックスやフレームの上に子コントロールを作成するときは、先にピクチャー ボックスやフレームを作成して、その中に子コントロールを配置する必要があります。フォームの上に配置したコントロールを後でピクチャー ボックスやフレームの中に移動させても、親子関係は成立しないので注意してください。

4.3　コマンド ボタンに対する応答

コマンド ボタンは通常、クリックすることによって処理を呼び出すために使われます。Caption プロパティを使ってボタンの上にテキストを表示すると、そのボタンがどのような働きをするのかが分かりやすくなります。

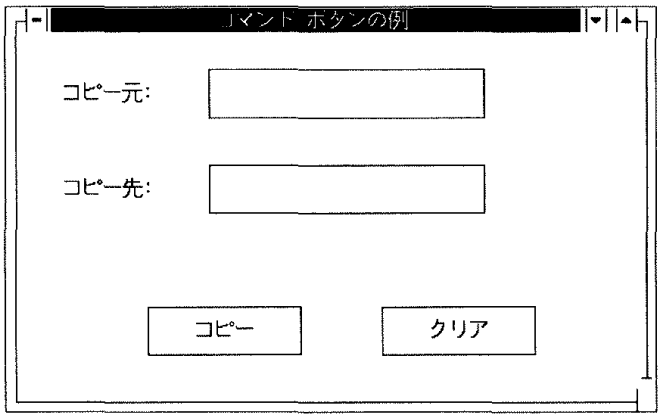
ユーザーがコマンド ボタンを選ぶには、次に示すような方法があります。

- マウスでボタンをクリックする。
- [TAB] キーを使ってそのボタンにフォーカスを移動した後、[SPACE] キーまたは [ENTER] キーを押す。
- そのコマンド ボタンがそのフォームのデフォルトのボタンであるとき (Default プロパティが True に設定されているとき) は、[ENTER] キーを押すだけでそのボタンをクリックしたことになる。
- そのコマンド ボタンがそのフォームの Cancel ボタンであるとき (Cancel プロパティが True に設定されているとき) は、[ESC] キーを押すだけでそのボタンをクリックしたことになる。

重 要 上に示した各種の操作は、同じ Click イベント プロシージャを呼び出します。このことは、ユーザーに対してマウスとキーボードの両方のインターフェイスを用意するときでも、対応するイベントプロシージャは 1 つでよいことを意味しています。

コマンド ボタンの例

次に示すアプリケーションは、2 つのコマンド ボタンを持っています。このうち <コピー> ボタンは、上のテキスト ボックス (txtSrc) から下のテキスト ボックス (txtDest) にテキストをコピーし、<クリア> ボタンは、2 つのテキスト ボックスの内容を消去します。



ユーザーがコピー元 (txtSrc) テキスト ボックスに文字列を入力した後、プログラムは次のように動作します。

- ユーザーが <コピー> ボタンを選んだときは、cmdDup_Click イベント プロシージャが文字列を大文字に変換して、コピー先テキスト ボックスにコピーします。
- ユーザーが <クリア> ボタンを選んだときは、cmdClearOut_Click プロシージャが 2 つのテキスト ボックスに null 文字列 ("") を代入して、その内容を消去します。

▼ 例題のアプリケーションを作成するには

1. 2 つのラベル, 2 つのテキスト ボックス, 2 つのコマンド ボタンをそれぞれ作成します。
2. 各コントロールのプロパティを、次のように設定します。

コントロール	プロパティの設定	使用するイベント
上段のラベル	Caption = コピー元:	
下段のラベル	Caption = コピー先:	
上段のテキスト ボックス	CtlName = txtSrc Text = lower case	
下段のテキスト ボックス	CtlName = txtDest Text = UPPER CASE	

左側のコマンド ボタン	CtlName = cmdDup Caption = コピー	Click
右側のコマンド ボタン	CtlName = cmdClearOut Caption = クリア	Click

3. <コピー> (cmdDup) コマンド ボタンの Click イベント プロシージャに、大文字への変換処理を加えます。具体的には、txtSrc テキスト ボックスの中身 (txtSrc.text) を **UCASE$** 関数で大文字に変換して txtDest.text に代入します。

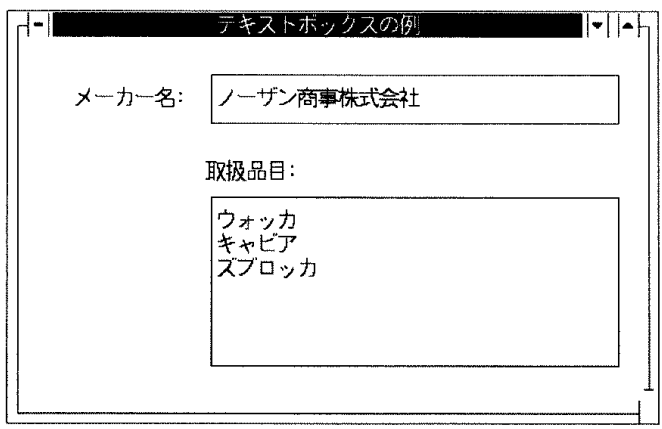
```
SUB cmdDup_Click()
    txtDest.Text = UCASE$(txtSrc.Text)
END SUB
```

4. テキスト ボックスの中身を消去する処理として、cmdClearOut_Click イベント プロシージャに次のコードを追加します。

```
SUB cmdClearOut_Click()
    txtSrc.Text = ""
    txtDest.Text = ""
END SUB
```

4.4　テキスト ボックスを使った入力の取得と出力の表示

テキスト ボックスは、テキストの入力と表示の両方の目的に使えます。



テキスト ボックスに表示されたテキストは、実行時にユーザーが変更できます。ユーザーが直接変更できないテキストを表示したいときは、ラベルを使います (ラベルについては、この章の「ラベルを使ったテキストの表示」を参照してください)。

第 2 章では、ユーザーに Hello, world というメッセージを表示するもっとも簡単なテキスト ボックスの例を示しました。この例では、コマンド ボタンがクリックされると、テキスト ボックスの Text プロパティに Hello, world という文字列が設定され、画面に表示されます。

テキスト ボックスの内容を読み出したり、表示したりするには、Text プロパティを使います。必要に応じて **VAL, STR$, FORMAT$** の各関数も利用します。

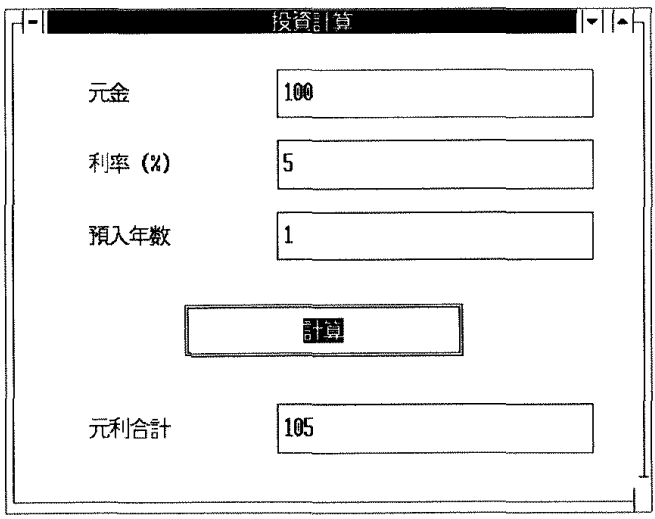
- Text プロパティは、テキスト ボックスに表示される文字列を示しています。ユーザーがテキスト ボックスに入力した文字列は、そのまま Text プロパティに代入されます。その他にも、フォーム デザイナのプロパティ バーを使ったり、コード中から代入することによっても、Text プロパティに文字列を設定できます。
- **VAL** 関数を使うと、数値を示す文字列を実際の数値に変換できます。この関数は、テキスト ボックスに入力された値で数値計算を行うときに使います。
- **STR$** と **FORMAT$** の 2 つの関数は、数値をテキスト ボックスに表示できる文字列に変換するのに使います。

テキスト ボックスには、最初からカット, コピー, ペーストの機能が組み込まれています。[CTRL] + [X] または [SHIFT] + [DEL] で切り抜き、[CTRL] + [C] または [CTRL] + [INS] で複写、[CTRL] + [V] または [SHFT] + [INS] で貼り付けが行えます。CLIPBOARD オブジェクトを使って特にプログラミングする必要はありません。

参 考 ユーザーがテキスト ボックスへの入力を行うときには、1 文字入力するごとに、そのテキスト ボックスに対する Change イベントが発生します。このためすべての文字の入力が完了してからテキスト ボックスのデータを処理しようとするときは、Change イベント プロシージャではなく、コマンド ボタンの Click プロシージャにコードを追加するか (次のプログラム例を参照)、テキスト ボックスの LostFocus プロシージャにコードを記述します。

投資計算のプログラム例

次に示す投資計算のプログラム例では、入力と出力の両方にテキスト ボックスを使っています。



ユーザーは元金, 利率(%), 預入年数の各テキスト ボックスに値を入力した後、コマンド ボタン Command1 をクリックします。ユーザーがコマンド ボタンをクリックすると、イベント プロシージャが次の処理を実行します。

1. ユーザーがテキスト ボックスに入力したテキストを取得します。
2. 数値計算で使えるように (**VAL** 関数を使って) 文字列を数値に変換した後、各変数に代入します。
3. 最終的な収支結果を計算し、得られた値を変数 EndBal に代入します。
4. 収支結果を文字列に変換し、フォーマットを整えてからテキスト ボックス txtEnd の Text プロパティに代入します。

▼ 例題のアプリケーションを作成するには

1. 4 つのテキスト ボックス, 4 つのラベル, 1 つのコマンド ボタンをそれぞれ作成します。
2. 各テキスト ボックスのプロパティを、次のように設定します。

コントロール	プロパティの設定	主なイベント
元金テキスト ボックス	CtlName = txtStart	
利率テキスト ボックス	CtlName = txtRate	
預入年数テキスト ボックス	CtlName = txtYears	
元利合計テキスト ボックス	CtlName = txtEnd	
コマンド ボタン	CtlName = Command1	Click

3. Command1_Click イベント プロシージャに、次のコードを追加します。

```
SUB Command1_Click()
    StartBal = VAL(txtStart.Text)          '元金の読み込み
    Rate = VAL(txtRate.Text)               '利率の読み込み
    Years = VAL(txtYears.Text)             '預入年数の読み込み
    EndBal = StartBal * (1 + Rate / 36500) ^ (Years * 365)
                                           '収支結果の計算
    txtEnd.Text = FORMAT$(EndBal, #,###,##0")
                                           '収支結果の表示
END SUB
```

FORMAT$** 関数を使うと、出力する文字列の書式を制御できます。**FORMAT$ 関数の出力書式の指定方法について詳しくは、オンライン ヘルプまたは「リファレンス」を参照してください。

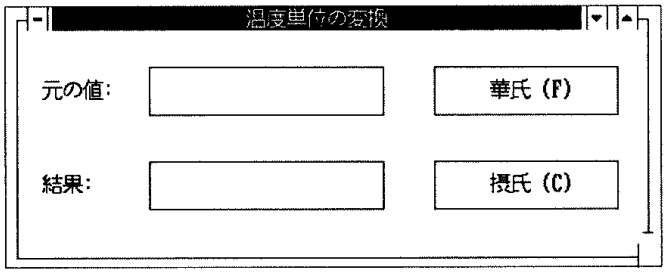
4.5 ラベルを使ったテキストの表示

ラベル コントロールは、ユーザーが直接には変更できないテキストを表示します。複数行のテキストの表示も可能です。ラベルはしばしば、テキスト ボックスやスクロール バーなどの Caption プロパティを持たないコントロールにタイトルを付ける目的で使われます。

ラベルの Caption プロパティで、ラベルに表示するテキストを設定できます。テキストに枠を付ける場合は、BorderStyle プロパティを使います。このプロパティに 1, 2 のいずれか (それぞれ細線, 太線に対応) を設定すると、ラベルに境界線 (枠線) が表示され、テキスト ボックスに似た外観になります。境界線付きのラベルは、アプリケーションでさまざまに利用することができます。

温度単位変換のプログラム例

次のアプリケーションでは、摂氏 (Celsius) と華氏 (Fahrenheit) の相互換算を行います。このプログラムでは、3 つのラベル, 1 つのテキスト ボックス, 2 つのコマンド ボタンを使います。初めにユーザーがテキスト ボックスに数値を入力し、次に 2 つあるコマンド ボタンのうちのどちらが押されたかに応じて華氏または摂氏への換算を実行します。換算結果は、テキスト ボックスと同様の外観を持つラベルに表示されます。このラベルは BorderStyle プロパティが 1 (細線) に設定されています。また、計算結果を Caption プロパティを使って表示しています。



ラベルの代わりにテキスト ボックスを使って変換後の温度を表示することもできますが、テキスト ボックスでは表示された情報をユーザーが変更できるので、このアプリケーションのような場合は適当ではありません。境界線付きのラベルはテキスト ボックスと外観は似ていますが、中に表示された情報をユーザーが直接修正することはできません。

▼ **例題のアプリケーションを作成するには**

1. フォーム上に 3 つのラベルと 1 つのテキスト ボックス、2 つのコマンド ボタンを配置します。
2. コントロールのプロパティを、次のように設定します。

オブジェクト	プロパティの設定	使用するイベント
フォーム	Caption = 温度単位の変換	
最初のラベル	Caption = 元の値:	
2 番目のラベル	Caption = 結果:	
3 番目のラベル	CtlName = lblOutputTemp Caption = (なし) BorderStyle = 1 (細線)	
上段のコマンド ボタン	CtlName = cmdToDegreesF Caption = 華氏 (F)	Click
下段のコマンド ボタン	CtlName = cmdToDegreesC Caption = 摂氏 (C)	Click
テキスト ボックス	CtlName = Text1 (デフォルト) Text = (なし)	

3. cmdToDegreesF_Click イベント プロシージャに、次のコードを追加します。

```
SUB cmdToDegreesF_Click ()                    '華氏温度への変換
    DegreesC = VAL(Text1.Text)                '摂氏温度の読み込み
    DegreesF = (DegreesC * 9/5) + 32          '華氏温度に変換
    lblOutputTemp.Caption = STR$(DegreesF)  'Captionとして表示
END SUB
```

4. cmdToDegreesC_Click イベント プロシージャに、次のコードを追加します。

```
SUB cmdToDegreesC_Click ()                  '摂氏温度への変換
    DegreesF = VAL(Text1.Text)              '華氏温度の読み込み
    DegreesC = (DegreesF - 32) * 5/9        '摂氏温度に変換
    lblOutputTemp.Caption = STR$(DegreesC)
                                            'Captionとして表示
END SUB
```

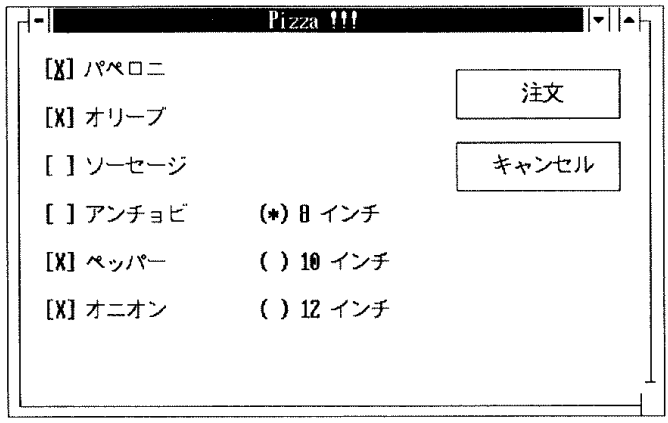
どちらのプロシージャにおいても、DegreesC と DegreesF が、それぞれ摂氏と華氏を示すローカル変数として使われている点に注意してください。これらの変数は明示的に宣言されていないため、そのデータ型はデフォルトである **SINGLE** になります。

4.6　オプション ボタンとチェック ボックスを使った選択項目の設定

アプリケーション上で、オプション ボタンまたはチェック ボタンを使ってオプションを設定することができます。

- オプション ボタンは、いくつかのオプション (項目) から 1 つを選ぶときに使います。同一のグループに属するオプション ボタンのうち、どれか 1 つだけが常にオンになっています。

- チェック ボックスは、さまざまな機能のオン / オフを設定するときに使います。複数のチェック ボックスを同時にオンすることができます。



これらのコントロールは、通常はダイアログ ボックスの中で使われます。上の例では、ピザのトッピングの種類をチェック ボックスで、また生地のサイズをオプション ボタンで、それぞれ選ぶようになっています。

オプション ボタンの利用

オプション ボタンについては、次のような特長があります。

- ボタンの右側に表示されるテキストは、Caption プロパティによって設定します。
- ボタンの状態は、Value プロパティによって決ります。-1 (True) のときは選択、0 (False) のときは非選択を意味します。
- フレームまたはピクチャー ボックスを使うと、オプション ボタンをいくつかのグループに分けることができます。各グループごとに 1 つのオプション ボタンが選択されます。

オプション ボタンのグループ化

フォームの上に直接置かれたオプション ボタンは、すべて同じグループに属します (フレームやピクチャー ボックスの中は別のグループである点に注意してください)。1 つのフォーム上で複数のオプション ボタンを組み合わせて使おうとするときは、フレームやピクチャー ボックスを使って、必要なグループ分けを行ってください。フレームやピクチャー ボックス内に配置されたオプション ボタンは、それぞれ独立したグループを形成します。これは、同じ親を持つすべてのオプション ボタンが 1 つのグループを形成することを意味しています。

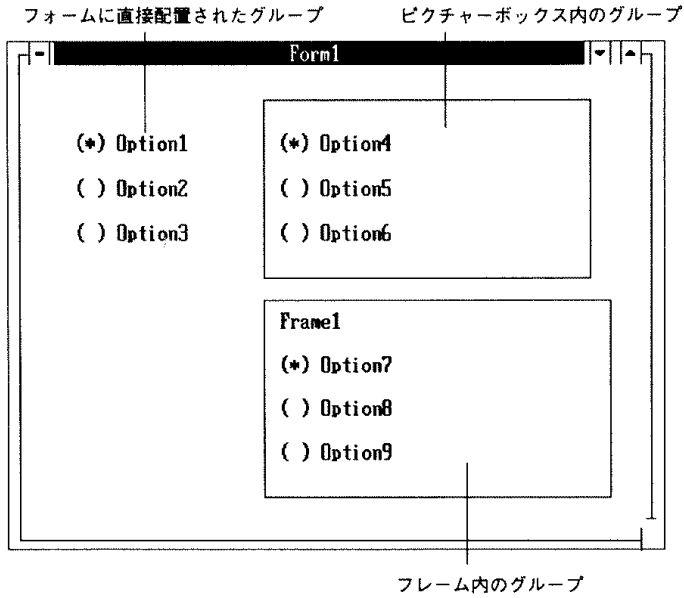


図 4.1　オプション ボタンのグループ化

フレームを使ったオプション ボタンのグループ化について詳しくは、この章の「フレームによるコントロールのグループ化」を参照してください。

オプション ボタンの選択と解除

選択されたオプション ボタンは、() の中にアスタリスク (*) が表示されます。オプション ボタンを選ぶには、次の方法があります。

- オプション ボタンをクリックする。
- [TAB] キーを使って希望のオプション ボタンのグループにフォーカスを移動し、方向キーを使ってオプション ボタンを選ぶ。
- コード中から、Value プロパティを True (-1) に設定する。

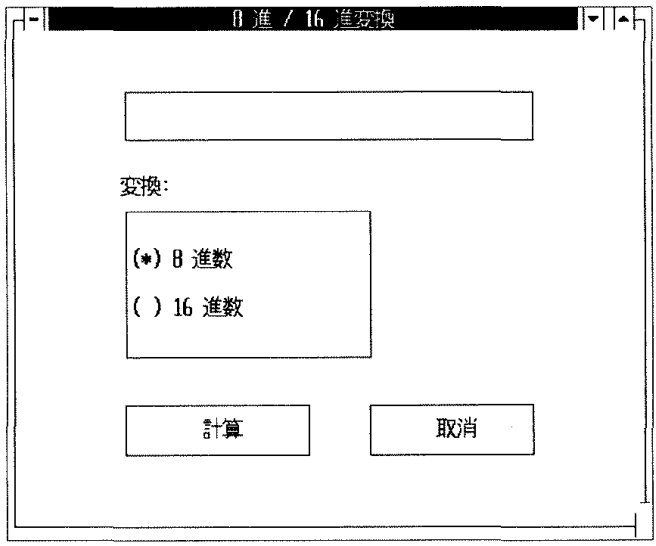
```
optAmexButton.Value = -1
```

アプリケーションの起動時に選択されているオプション ボタンは、フォーム デザイナで最後に操作したものとなります。コード中から設定することもできます。

特定のオプション ボタンを選択できないようにするには、Enabled プロパティを False (0)に設定します。こうすると、実行時にボタンが選択不能 (薄い色で表示) になります。一般的には、必要に応じてプログラム コードで Enabled プロパティの設定を切り換えます。

基数変換のプログラム例

次に示すプログラム例では、10 進数の数値を 16 進数または 8 進数に変換します。変換する基数はオプション ボタンで選びます。このプログラムでは、ユーザーがコマンド ボタン <計算> をクリックすると、テキスト ボックスに表示されている数字が指定した基数表示の値に変化します。



▼ このアプリケーションを作成するには

1. フォームとコントロールを作成します。
2. 各コントロールのプロパティを、次のように設定します。

コントロール	プロパティの設定	使用するイベント
下段のオプション ボタン	CtlName = optHexButton Caption = 16 進数	
上段のオプション ボタン	CtlName = optOctButton Caption = 8 進数	
テキスト ボックス	CtlName = Text1 Text = (なし)	Change
左側のコマンド ボタン	CtlName = Command1 Caption = 計算	Click
右側のコマンド ボタン	CtlName = Command2 Caption = 取消	Click

3. フォーム モジュールのモジュールレベルで、CurrentNum という名前の変数を初期化します。

```
DIM SHARED CurrentNum
```

この変数は、Text1 の値を数値として保持します。**SHARED** というキーワードを指定することによって、このフォーム モジュール内に存在するすべてのプロシージャからこの変数にアクセスできるようになります。

変数はデフォルトで 0 に初期化されるため、ここではこれ以外のコードを記述して初期値を代入する必要はありません。

4. Text1_Change イベント プロシージャに、次のコードを追加します。

```
SUB Text1_Change ()
    CurrentNum = VAL(Text1.Text)
END SUB
```

Text1_Change イベント プロシージャは、テキスト ボックスに入力された文字列を数値に変換して、変数 CurrentNum に代入します。

文字列から数値への変換には **VAL** 関数を使います。**VAL** 関数は 8 進数, 10 進数, 16 進数のいずれで表記された文字列でも扱えます。8 進数は &o, 16 進数は &h を数字の前に付けて表記します。

5. Command1_Click イベント プロシージャに、次のコードを追加します。

```
SUB Command1_Click ()
    IF optHexButton.Value = 1 THEN
                        '16 進数のボタンが選択されている場合
        Text1.Text = HEX$(CurrentNum)        '16 進数で表示
    ELSE                '8 進数のボタンが選択されている場合
        Text1.Text = OCT$(CurrentNum)        '8 進数で表示
    END IF
END SUB
```

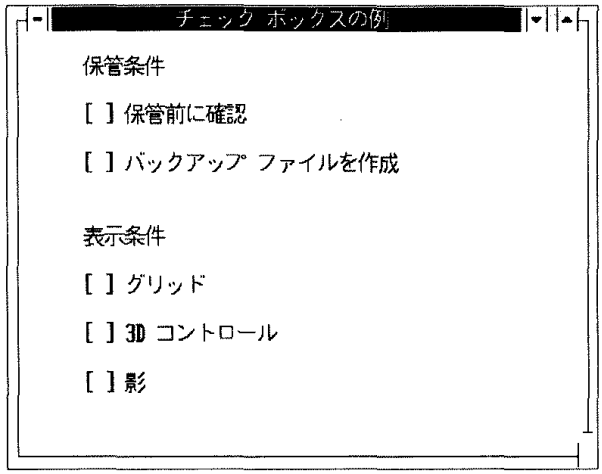
Click イベント プロシージャは、オプション ボタンの状態に応じて 8 進数と 16 進数のどちらが選択されているかを判断します。**OCT$** と **HEX$** の 2 つの関数は、入力された値 (CurrentNum) をそれぞれ 8 進数, 16 進数に変換します。

6. Command2_Click イベント プロシージャに、**END** ステートメントを追加します。

```
SUB Command2_Click ()
    SCREEN.HIDE
    END
END SUB
```

チェック ボックスの利用

チェック ボックスは、さまざまな機能をオン / オフするために使用します。チェック ボックスは独立したコントロールで、複数を同時にオンにすることもできます。フォーム上に分かりやすくレイアウトするために、フレームやピクチャー ボックスの内部に配置してグループに分けることもできますが、他のチェック ボックスの設定に影響を受けることはありません。



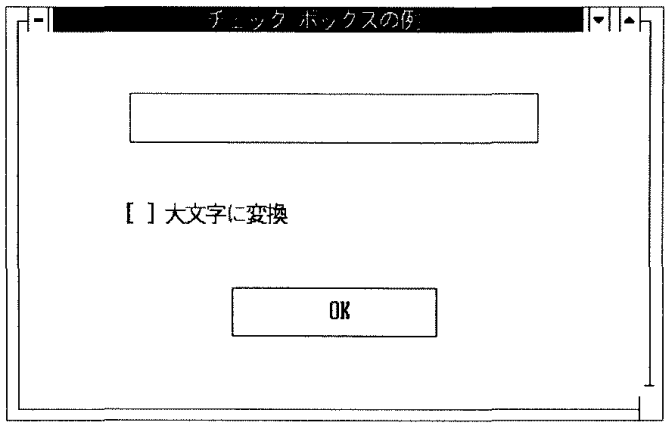
チェック ボックスはオプション ボタンと異なり、それぞれが独立して機能するため、複数のチェック ボックスを同時にオンにすることができます。

チェック ボックスでの設定をアプリケーションの動作に反映するには、Value プロパティの設定値を調べます。アプリケーション上でチェック ボックスをクリックすると Value プロパティの値が 0 と 1 の間で交互に切り替わります。チェック ボックスが操作されていないことを示すために、初期設定値を 2 にすることもできます。この場合は、フォーム デザイナでの設定や、プログラム コードからの操作によって Value プロパティを 2 に設定します。マウスのクリックによって Value プロパティに 2 がセットされることはありません。

Value プロパティの設定	チェック ボックスの状態
0	未チェック (オフ)
1	チェック (オン)。チェック ボックスに X が表示されます。
2	チェック ボックスがまだ操作されていないことを示します (この設定ができるのは、フォーム デザイナかコード内のいずれかに限られます。実行時にユーザーがこの設定にすることはできません)。チェック ボックスに - が表示されます。

小文字大文字変換のプログラム例

次に示すプログラム例では、コマンド ボタンをクリックすると、チェック ボックスがオンになっているときだけテキスト ボックス内の文字を大文字に変換します。このアプリケーションでは、チェック ボックス, テキスト ボックス, コマンド ボタンをそれぞれ 1 つずつ使っています。



コマンド ボタンがクリックされると、Command1_Click イベント プロシージャはチェック ボックスがチェックされているかどうか (Value プロパティが 1 かどうか) を調べます。チェックされているときだけ、テキスト ボックスに入力されている文字列を大文字に変換します。

▼ このアプリケーションを作成するには

1. テキスト ボックス, チェック ボックス, コマンド ボタンを作成します。
2. プロパティを次のように設定します。

コントロール	プロパティの設定	使用するイベント
テキスト ボックス	CtlName = Text1 Text = (なし)	
チェック ボックス	CtlName = Check1 Caption = 大文字に変換	
コマンド ボタン	CtlName = Command1 Caption = OK	Click

3. Command1_Click イベント プロシージャに、次のコードを追加します。

```
SUB Command1_Click ()
    IF Check1.Value = 1 THEN
                        'チェック ボックスがチェックされている場合
        Text1.Text = UCASE$(Text1.Text)
    END IF
END SUB
```

UCASE$ 関数が、テキスト ボックスのテキストを大文字に変換します。

4.7 リスト ボックスとコンボ ボックスによる値の設定

リスト ボックス コントロールは、選択項目のリストを表示します。

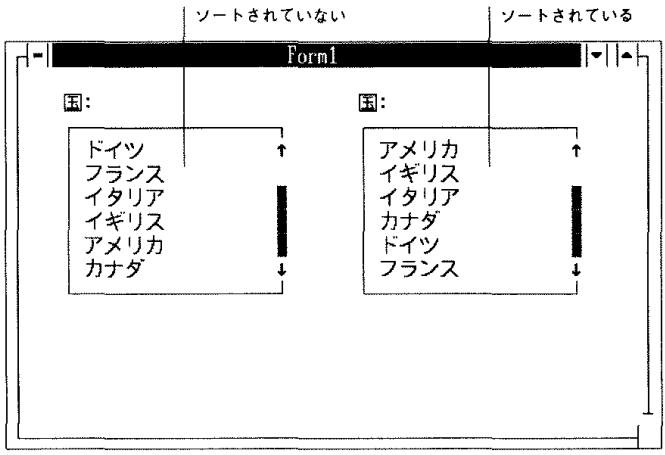


図 4.2 リスト ボックス

リスト ボックスはフォーム デザイナで作成しますが、リストされる項目はコード中から設定します。リストの中から 1 つを選択するときは、その名前を直接クリックするか、方向キーを使って反転表示させます。

参 考 ファイル, ディレクトリ, ドライブの各リスト ボックスの詳細については、第 10 章「ファイルシステム コントロールの利用」を参照してください。

コンボ ボックス コントロールは、テキスト ボックスとリスト ボックスの特性を合わせ持っています。コンボ ボックスは、Style プロパティの設定によって、ドロップダウン コンボ ボックス、シンプル コンボ ボックス、ドロップダウン リスト ボックスのいずれかになります。

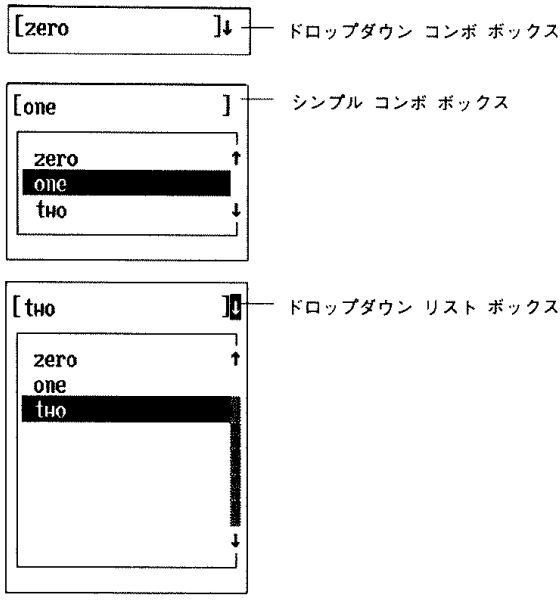


図 4.3　コンボ ボックス

表 4.2 に、コンボ ボックスの種類と、対応する Style プロパティの設定を示します。

表 4.2　コンボ ボックスの種類

種類	Style プロパティの設定	機能
ドロップダウン コンボ ボックス	0	ユーザーは、テキスト ボックス部分に直接入力することも、ドロップダウンするリスト ボックス部分から項目を選択することもできます。右側の下向矢印をクリックするか、[メニュー]+[↓] キーを押すと、リスト ボックス部分がドロップダウンします。また、[↑] キーと [↓] キーでも項目を選べます。フォーカスが他のコントロールへ移るか、[ENTER] または [ESC] キーを押すと、リスト ボックス部分はクローズします。
シンプル コンボ ボックス	1	リストが常に表示されている点を除いて、ドロップダウン コンボ ボックスと同じです。
ドロップダウン リスト ボックス	2	ユーザーはリスト ボックス部分から項目を選択するだけで、テキスト ボックス部分に直接入力できないことを除いて、ドロップダウン コンボ ボックスと同じです。

リスト ボックスではなくコンボ ボックス (Style = 0 または 1) を使う利点は、既存の選択対象から選ぶだけでなく、ユーザーがキー入力によって値を設定できるという点にあります。そのときリスト中で反転表示されている項目は、テキスト ボックス部分にも表示されています。限られた画面の中で、より多くのリストから項目を選べるようにしたいときは、ドロップダウン リスト ボックス (Style = 2) を使います。

表 4.3 に、リスト ボックスとコンボ ボックスの主なイベント, プロパティ, メソッドをまとめて示します。

表 4.3 リスト ボックスとコンボ ボックスのまとめ

イベント, プロパティ, メソッド	機能
ADDITEM メソッド	リスト ボックスまたはコンボ ボックスに項目を追加します。
REMOVEITEM メソッド	リスト ボックスまたはコンボ ボックスから項目を削除します。
Click および DblClick イベント	ユーザーがリストから項目を選び、確定すると発生します。
Change イベント	コンボ ボックスのテキスト ボックス部分に何か変更が加えられたときに発生します。ドロップダウン リスト ボックスでは発生しません。
Text プロパティ	そのとき選択されているリスト項目 (文字列) を示します。
List プロパティ	リストの全項目を含む文字列配列を示します。
ListIndex プロパティ	現在選択されている項目の通し番号を示します。先頭の項目が 0 になります。このプロパティを読み出すことによって、現在何番目の項目が選択されているのかが分かります。また、このプロパティに値を代入することによって選択されている項目を切り換えることができます。
ListCount プロパティ	リスト内の項目の個数を示します。
Sorted プロパティ	リストまたはコンボ ボックスの要素をアルファベット順にソートするかどうかを示します。フォーム デザイナでのみ設定できます。

これらの機能については、これから順に説明します。特に記述がない限り、すべてのプロシージャはリスト ボックスとコンボ ボックスの両方に適用されます。

リストへの項目の登録

リスト ボックスやコンボ ボックスに項目を追加するには、**ADDITEM** メソッドを使います。**ADDITEM** メソッドの構文は、次のとおりです。

box.**ADDITEM** *item* [, *index*]

引数	説明
box	リスト ボックスまたはコンボ ボックスの名前 (CtlName)
item	リストに追加する項目を示す文字列
index	新しい項目が、リスト内のどの位置に挿入されるかを示す番号。*index* が 0 のときにリストの先頭に挿入されます。省略すると、項目はリストの最後 (またはソートされた順番に従った適切な位置) に挿入されます。

ADDITEM メソッドはどのような場合にも利用できるので、ユーザーの操作に応答してリストに項目を追加することが可能です。

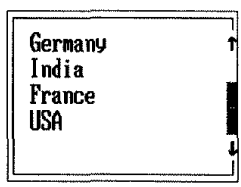
リスト ボックスやコンボ ボックスに数値を追加するには、最初に数値を文字列に変換する必要があります。**STR$** または **FORMAT$** 関数を使ってください。

リスト ボックスの例

次のプログラム例では、Germany, India, France, USA の各項目を lstPopulations という名前のリスト ボックスに追加しています。

```
SUB Form_Load ()
    lstPopulations.AddItem "Germany"
    lstPopulations.AddItem "India"
    lstPopulations.AddItem "France"
    lstPopulations.AddItem "USA"
END SUB
```

実行時には、次のように表示されます。

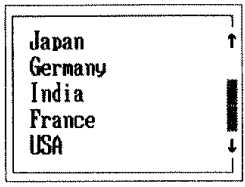


指定位置への項目の追加

新しい項目をリスト内の指定した位置に追加するには、**ADDITEM** メソッドで項目の後に位置を示す整数を指定します。次の例では、リストの先頭に Japan が挿入され、他の項目が 1 つずつ下に下がります。

```
    lstPopulations.AddItem "Japan", 0
```

リストの先頭は 1 ではなく 0 であることに注意してください。



リスト ボックスの Sorted プロパティを True に設定しておけば、リスト内の項目は常にソートされた状態になります。このとき、アルファベットは大文字小文字を区別せずにソートされ、2 バイト文字は文字コード順にソートされます。**ADDITEM** を使って項目を追加すると、新しい項目は該当する位置に挿入されます。Sorted プロパティはフォーム デザイナでしか設定できません。また、Sorted プロパティを True (-1) に設定すると、**ADDITEM** で項目を追加するときに *index* の指定ができません。

注 意 Sorted プロパティが True (-1) に設定されているときに、**ADDITEM** メソッドの *index* 引数を指定した場合、動作は保証されません。

リストからの項目の削除

リスト ボックスやコンボ ボックスから項目を削除するには、**REMOVEITEM** メソッドを使います。このとき、引数 *index* に、削除する項目の先頭からの位置を示す整数を指定します。

box.**REMOVEITEM** *index*

box と *index* の意味は、**ADDITEM** メソッドと同じです。

例えばリストの先頭の項目を削除するには、次のコードをプログラムに追加します。

```
lstPopulations.REMOVEITEM 0
```

リスト項目の操作

実行時にリスト ボックスおよびコンボ ボックスの内容を操作するには、次に示すプロパティを使います。

プロパティ	説明
Text	そのとき選択されている項目 (文字列)を示します。
List	リストの全項目を含む文字列配列を示します。
ListIndex	そのとき選択されている項目のリスト中の位置を示します。
ListCount	リスト内の項目数を示します。

Text プロパティの利用

そのとき選択 (反転表示) されている項目の値を最も簡単に取得するには、Text プロパティを使います。

- リスト ボックスの Text プロパティは、常にそのとき選択 (反転表示) されている項目（の名前）を示しています。
- コンボ ボックスの Text プロパティは、コントロールのテキスト ボックスの部分に表示されている文字列を示しています。これは、リストから選択された項目か、ユーザーがテキスト ボックス部分に入力した文字列のどちらかです。

例えば、次のようなコードを記述すると、ユーザーが Population リスト ボックスから USA を選んだときに、アメリカ合衆国に関する情報が表示されます。

```
SUB lstPopulation_Click()
    IF lstPopulations.Text = "USA" THEN
        MSGBOX "アメリカの人口は 2 億 5 千万です。"
    END IF
END SUB
```

lstPopulation リスト ボックスのText プロパティには、そのとき選択 (反転表示) されている項目の項目名が設定されています。プログラムは "USA" が選択されているかどうかをチェックして、選択されていればアメリカの人口に関するダイアログ ボックスを表示します (リスト ボックスの Click イベントについては、この章の「リスト ボックスのイベント」で説明します)。

List プロパティの利用

List プロパティを使えば、リスト中の全項目にアクセスできます。このプロパティは、リスト中のすべての項目名を含む配列になっています。各項目は文字列として登録されており、リスト内の項目を参照するには次の構文を使います。

box.**List** (*index*%)

この構文で、引数 *box* は参照するリスト ボックスまたはコンボ ボックス (の CtlName) を指し、*index* は項目の位置を示します。リストの先頭の項目の *index* は 0 であり、以下順番に 1, 2, 3 と番号が付けられます。例えば次の構文は、リスト ボックスの 3 番目の項目 (*index* = 2) を変数に代入します。

```
A$ = lstPopulations.List(2)
```

ListIndex プロパティの利用

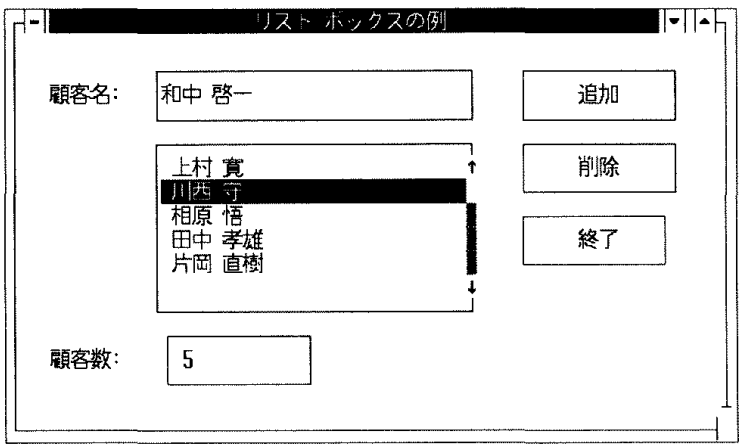
そのとき選択 (反転表示) されている項目の *index* を知りたいときは、ListIndex プロパティを使います。ListIndex プロパティはそのとき選択 (反転表示) されている項目の *index* を示しています。このプロパティを利用できるのは実行時だけで、フォーム デザイナで設定することはできません。

リストの先頭の項目が選ばれているときは ListIndex プロパティは 0 になり、以下順に 1 以降の番号が割り当てられます。

参 考 項目が選択されていなかったり、ユーザーがリストから既存の項目を選択したのではなくコンボ ボックス (Style 0 または 1) に項目を入力した場合は、ListIndex は -1 に設定されます。

ListIndex と ListCount のプログラム例

次に示すプログラムでは、**ADDITEM** と **REMOVEITEM** の 2 つのメソッドを ListIndex と ListCount の各プロパティと組み合わせて使うことによって、アプリケーション上でリストの項目の追加および削除を行っています。具体的には、ユーザーがテキスト ボックスに新しい項目名を入力し、<追加> ボタンをクリックすると、項目名がリスト ボックスに追加されます。逆にリストから項目を選んで <削除> ボタンを押すと、その項目がリスト ボックスから削除されます。



2 番目のテキスト ボックスにはリスト ボックスに含まれる項目の総数が表示されていて、項目の追加または削除が行われるたびに値が更新されます。

▼ このアプリケーションを作成するには

1. 上の図に従って、1 つのテキスト ボックス, 1 つのリスト ボックス, 3 つのラベル, 3 つのコマンド ボタンを含むフォームを作成します。
2. 各コントロールのプロパティを次のように設定します。

コントロール	プロパティの設定	主なイベント
テキスト ボックス	CtlName = Text1 Text = (なし)	
上段のラベル	CtlName = Label1 Caption = 顧客名:	
下段のラベル	CtlName = Label2 Caption = 顧客数:	
項目数表示用のラベル	CtlName = lblClients Caption = (なし) BorderStyle = 1 (細線)	

リスト ボックス	CtlName = List1 Sorted = True (-1)	
上段のコマンド ボタン	CtlName = cmdAdd Caption = 追加	Click
中段のコマンド ボタン	CtlName = cmdRemove Caption = 削除	Click
下段のコマンド ボタン	CtlName = cmdExit Caption = 終了	Click

3. cmdAdd_Click イベント プロシージャとして、次のコードを記述します。

```
SUB cmdAdd_Click()
    Client$ = Text1.Text      '顧客名を読み込んで Client$ に代入
    List1.ADDITEM Client$     'リスト ボックスに追加
    Text1.Text = ""           'テキスト ボックスをクリア
    lblClients.Caption = STR$(List1.ListCount) '顧客数を表示
END SUB
```

4. cmdRemove_Click イベント プロシージャとして、次のコードを記述します。

```
SUB cmdRemove_Click()
    Ind% = List1.ListIndex
                '現在選択されている項目のインデックス番号を読み込む
    IF Ind% >= 0 THEN            '項目が選択されているかどうかを確認
        List1.REMOVEITEM Ind%    '項目をリスト ボックスから削除
        lblClients.Caption = STR$(List1.ListCount)
                                          '顧客数表示を更新
    ELSE
        BEEP
    END IF
END SUB
```

5. cmdExit_Click イベント プロシージャとして、次のコードを記述します。

```
SUB cmdExit_Click()
    SCREEN.HIDE
    END      'プログラムの終了
END SUB
```

ListCount プロパティの利用

リスト ボックスまたはコンボ ボックスの項目数を取得するには、ListCount プロパティを使います。このプロパティが利用できるのは、実行時に限られます。例えば、次に示すコードでは、リスト ボックスの項目数を参照するために ListCount プロパティを使っています。

```
A% = List1.ListCount
```

リスト ボックスのイベント

リスト ボックスの主なイベントには、Click と DblClick の 2 つがあります。

イベント	説明
Click	ユーザーがリスト ボックスの項目を選択するたびに発生します。選択には、リスト上で項目をクリックしたり、矢印キーを使って反転表示させたり、項目名の最初の文字をキーボードから入力するなどの方法があります。
DblClick	ユーザーがリスト ボックスの項目をダブルクリックしたときに発生します。

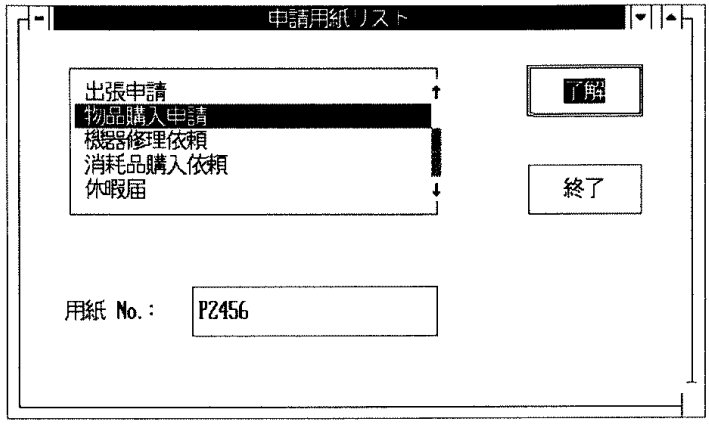
注 意 キーボード操作では、DblClick イベントは発生しません。

リスト ボックスのイベント (特にリスト ボックスがダイアログ ボックスの一部として表示されるときの) は、コマンド ボタンと併用する形で使い方を覚えるとよいでしょう。前に示したプログラムでは、リスト ボックスで 1 つの項目を選んでからコマンド ボタンをクリックすることによって、リストから選択された項目に対して何らかの操作を実行しています。

次に示す例では、リスト内の項目のダブルクリックが、項目を選択してからコマンド ボタンをクリックするという 2 つの操作に相当します。これは、リスト ボックスの DblClick イベント プロシージャが、コマンド ボタンの Click イベント プロシージャを呼び出すことで実現できます。コマンド ボタンの Click イベント プロシージャを呼び出すには、CALLステートメントを使う一般的な方法の他に、(リスト ボックスの DblClick イベント プロシージャの中で) コマンド ボタンの Value プロパティに -1 を代入する方法があります。ダブルクリックによって呼び出されるイベント プロシージャから、コマンド ボタンに対する イベント プロシージャを呼び出すことで、マウス ユーザーはすばやく操作ができ、またキーボード ユーザーも不都合なく操作できるようになります。

リスト ボックスのイベントのプログラム例

次に示すプログラム例では、ユーザーがリストから項目を選択してコマンド ボタンを押したときと、同じ項目をダブルクリックしたときで、同じ動作を実行するようになっています。ユーザーが、さまざまな業務が表示されているリスト ボックスから項目を選ぶと、テキスト ボックスに必要な用紙の番号が表示されます。



▼ このアプリケーションを作成するには

1. 上の図に従って、リスト ボックス、2 つのラベル、2 つのコマンド ボタンを持つフォームを作成します。

2. 各コントロールのプロパティを、次のように設定します。

オブジェクト	プロパティの設定	主なイベント
フォーム	FormName = Form1 Caption = 申請用紙リスト	Load
リスト ボックス	CtlName = List1	DblClick
左のラベル	CtlName = Label1 Caption = 用紙 No.	
右のラベル	CtlName = Label2 Caption = (なし) BorderStyle = 1 (細線)	
上側のコマンド ボタン	CtlName = Command1 Caption = 了解	Click
下側のコマンド ボタン	CtlName = Command2 Caption = 終了	Click

3. Form_Load イベント プロシージャでは、次のようにリスト ボックスを初期化します。

```
SUB Form_Load()
    List1.Additem "出張申請"
    List1.Additem "物品購入申請"
    List1.Additem "機器修理依頼"
    List1.Additem "消耗品購入依頼"
    List1.Additem "休暇届"
END SUB
```

4. Command1_Click イベント プロシージャとして、次のコードを記述します。

```
Command1_Click()
    SELECT CASE List1.Text
                'リスト ボックスで選択されている項目の文字列を読み込む
        CASE "出張申請"
            Text1.Text = "S4542"
        CASE "物品購入申請"
            Text1.Text = "P2456"
        CASE "機器修理依頼"
            Text1.text = "H8292"
        CASE "消耗品購入依頼"
            Text1.text = "V8294"
        CASE "休暇届"
            Text1.text = "E9921"
    END SELECT
END SUB
```

5. List1_DblClick イベント プロシージャから Command1_Click イベント プロシージャを呼び出すようにします。

```
SUB List1_DblClick()
    CALL Command1_Click
END SUB
```

または、

Command1 コマンド ボタンの Value プロパティに True (-1) を代入します。これによって、Command1_Click イベント プロシージャが自動的に呼び出されます。

```
SUB List1_DblClick()
    Command1.Value = -1
END SUB
```

6. Command2_Click イベント プロシージャに終了処理を記述します。

```
SUB Command2_Click()
    SCREEN.HIDE
    END
END SUB
```

コンボ ボックスのイベント

コンボ ボックスの主なイベントには、Change, Click, DblClick, DropDown の 4 つがあります。

イベント	説明
Change	ユーザーがコンボ ボックスのテキスト ボックス部分にデータを入力するか、コード内で Text プロパティを変更したときに発生します。コンボ ボックスの Style プロパティは、0 (ドロップダウン コンボ) または 1 (シンプル コンボ) に設定されている必要があります。コンボ ボックスの Style プロパティが 2 に設定されているとき (ドロップダウン リスト ボックス) は、テキスト ボックス部分が存在しないため、Change イベントは発生しません。
Click	ユーザーがコンボ ボックスのリスト ボックス部分から項目を選択したときに発生します。
DblClick	ユーザーがシンプル コンボ ボックス (Style = 1) の項目をダブルクリックしたときに発生します。シンプル コンボ ボックスでは DblClick イベントが発生します。Style プロパティが 0 または 2 に設定されている場合は、このイベントは発生しません。
DropDown	コンボ ボックスのリスト ボックス部分がドロップダウンするときに発生します。シンプル コンボ ボックス (Style = 1) の場合は、このイベントは発生しません。

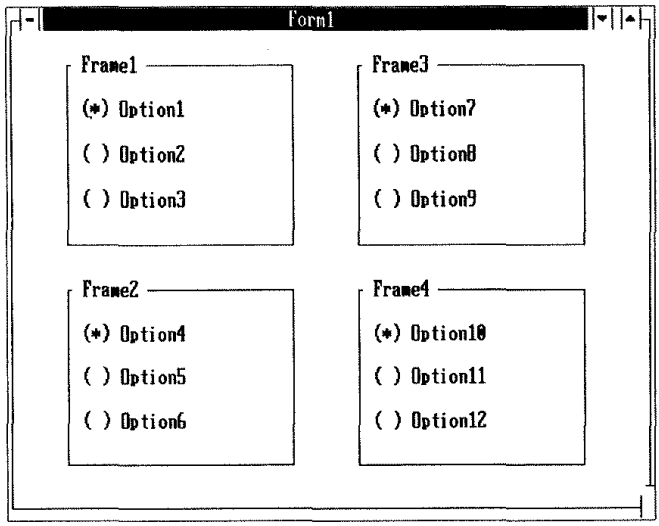
コンボ ボックスの Style プロパティが 0 または 2 に設定されているときは、リスト ボックス部分がドロップダウンするたびに、DropDown イベントが発生します。コンボ ボックスの右側の矢印をクリックするか、コンボ ボックスがフォーカスを持っているときに [メニュー] + [↓] キーを押すと、リスト ボックス部分がドロップダウンします。リストが表示される前にその内容を変更したいときは、DropDown イベント プロシージャにコードを記述します。

コンボ ボックスのイベントでサポートされないものを次の表にまとめます。

Style プロパティ	サポートされないイベント
0 (ドロップダウン コンボ)	DblClick
1 (シンプル コンボ)	DropDown
2 (ドロップダウン リスト)	Change, DblClick

4.8 フレームによるコントロールのグループ化

フレームを使うと、コントロールを視覚的にまとめて表示したり、オプション ボタンをグループ化したりできます。オプション ボタンをフレームによってグループ化すると、各グループで 1 つずつのオプション ボタンが選べるようになります。



▼ フレームを使ってコントロールをグループ化するには

1. フォーム デザイナで、フォーム上にフレームを配置します。
2. フレーム内にコントロールを配置します。

先にフレームを描き、後でその上にコントロールを追加したときには、フレームを移動するとその上のコントロールも合わせて移動します。一方、すでに配置されているコントロールをフレーム上に移動しても、コントロールはグループ化されず、フレームと一緒に移動しません。

フォームデザイナで、ツール ボックスをダブルクリックすると、デフォルトのサイズのコントロールがフォームの中央に配置されます。このようにして配置したコントロールをフレームやピクチャー ボックス内に移動しても、グループ化されないので注意してください。グループ化したいときは、フレームやピクチャー ボックス内でマウントをドラッグし、直接配置しなければなりません。

4.9　ピクチャー ボックスの利用

ピクチャー ボックスには、1 行または複数行のテキストを表示できます。テキストを表示するには、**PRINT** メソッドを使います。ピクチャー ボックスに表示されたテキストは、テキスト ボックスのようにユーザーが直接修正することはできません。フレームと同様に、ピクチャー ボックスでもコントロールのグループ化が可能です。

注　意　ピクチャー ボックスに、グラフィックスを表示することはできません。

ピクチャー ボックスのプログラム例

ピクチャー ボックスはフレームと異なり、フォーカスを持つことができます。

次の例は、ASCII 文字を使ってマウスでピクチャー ボックスに絵を書くプログラムです。

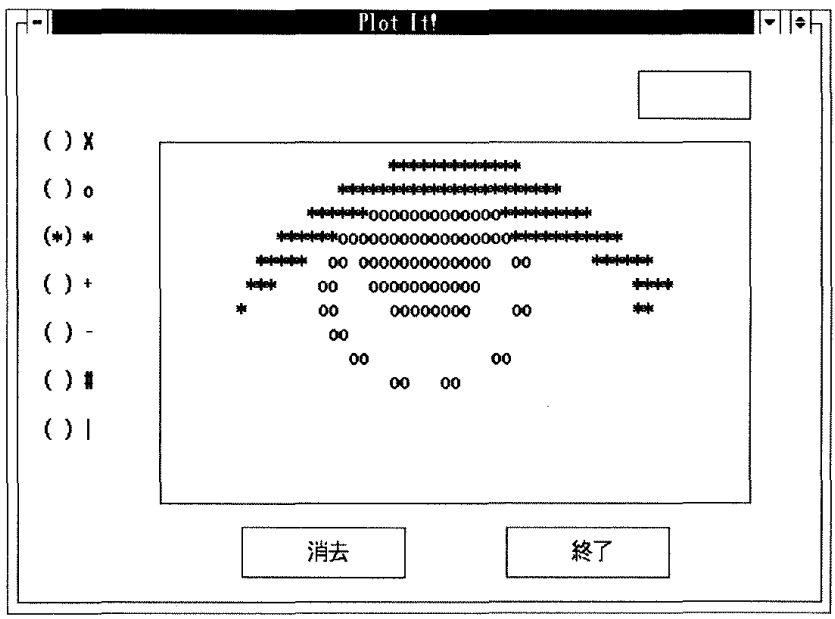


図 4.4　ASCII 文字のピクチャー ボックスへの出力例

ユーザーはオプション ボタンで描画に使うキャラクタを選び、ピクチャー ボックスの中にプロットします。フォームの右上のラベルには、マウス カーソルの現在の位置が数値で表示されます。

このプログラム例では、MouseDown イベント プロシージャを使ってピクチャー ボックスの中にキャラクタを表示しています。MouseDown イベントは、ユーザーがマウス ボタンを押すと常に発生します。MouseDown をはじめとする各種のマウスのイベントについては、第 13 章「マウス, キーボード, ドラッグ & ドロップ イベント」を参照してください。

▼ このアプリケーションを作成するには

1. フォーム デザイナで、インターフェイスを構成するフォームとコントロールを作成します。オプション ボタンはコントロール配列として作成します (作成の手順については、この章の後半の「コントロール配列の利用」を参照してください)。

2. 各コントロールのプロパティを、次のように設定します。

コントロール	プロパティの設定	使用するイベント
オプション ボタン(0)	CtlName = Option1 Caption = X Value = True	
オプション ボタン(1)	CtlName = Option1 Caption = o Value = False	
オプション ボタン(2)	CtlName = Option1 Caption = * Value = False	
オプション ボタン(3)	CtlName = Option1 Caption = + Value = False	
オプション ボタン(4)	CtlName = Option1 Caption =- Value = False	
オプション ボタン(5)	CtlName = Option1 Caption = # Value = False	
オプション ボタン(6)	CtlName = Option1 Caption = | Value = False	
ラベル	CtlName = Label1 Caption = (なし) BorderStyle = 1 (細線)	
ピクチャー ボックス	CtlName = Picture1 AutoRedraw = True	MouseDown
コマンド ボタン (消去)	CtlName = cmdErase Caption = 消去	Click
コマンド ボタン (終了)	CtlName = cmdExit Caption = 終了	Click

3. プログラミング環境に戻り、フォーム モジュールのモジュールレベルで次に示す定数を定義します。

```
CONST TRUE = -1
CONST FALSE = 0
```

4. ピクチャー ボックスのMouseDown イベント プロシージャに次のコードを記述します。

```
SUB Picture1_MouseDown (Button AS INTEGER, Shift AS INTEGER,
  ⇒ X AS SINGLE, Y AS SINGLE)
DIM Symbol as STRING
    FOR i = 0 to 6        'オプション ボタンの設定を 1 つずつ調べる
        IF Option1(i).Value = TRUE THEN
                          'オプション ボタンが選択されている場合
            Symbol = Option1(i).Caption
                          '選択されているキャラクタを Symbol に代入
        END IF
    NEXT
    '現在のマウスの位置を CurrentX と CurrentY に代入
    Picture1.CurrentX = X
    Picture1.CurrentY = Y
    Picture1.Print Symbol 'キャラクタをピクチャー ボックスに書く
    'CurrentX と CurrentY をラベルに表示
    Label1.Caption = FORMAT$(X,"##") + ", "
      ⇒ + FORMAT$(Y,"##")
END SUB
```

5. cmdErase_Click イベント プロシージャに次のコードを記述します。

```
SUB cmdErase_Click()
    Picture1.CLS 'ピクチャー ボックスをクリア
END SUB
```

6. cmdExit_Click イベント プロシージャに次のコードを記述します。

```
SUB cmdExit_Click()
    SCREEN.HIDE
    END      'プログラムを終了
END SUB
```

4.10　スクロール バーによる入力の制御

一般にスクロール バーは、長いリストや大量の情報を限られたスペースの中に表示するときに使われます。リストの中の現在位置はスクロール ボックスで示されます。スクロール バー コントロールは他のコントロールから独立して操作でき、独自のイベント, プロパティ, メソッドを持っています。テキスト ボックス, リスト ボックス, コンボボックスなど一部のコントロールには組み込み型のスクロール バーがありますが、ここで説明するスクロール バー コントロールとは異なります。

参　考　テキスト ボックスに組み込まれたスクロール バーを利用するかどうかを決るには、テキスト ボックスの ScrollBars プロパティを使います。

スクロール バー コントロールは、他のコントロールから独立してアプリケーションのインターフェイスとして使えます。例えば、ボリュームの設定や消費時間の表示などに利用できます。

図 4.5 に示すように、フォーム デザイナでフォーム上にスクロール バー コントロールを配置します。

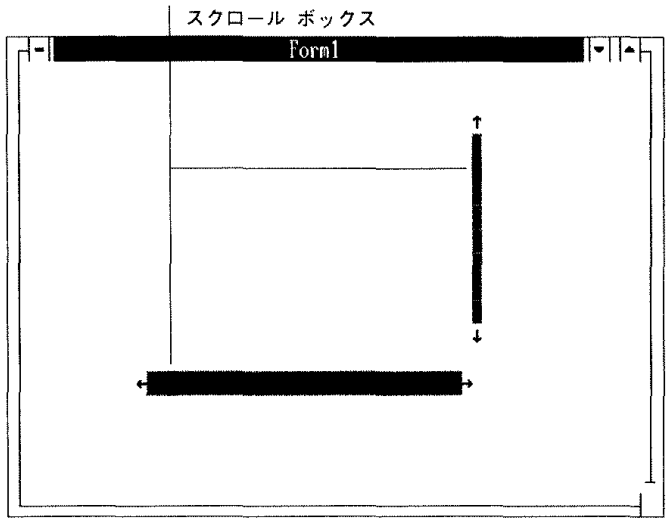


図 4.5 スクロール バー

スクロール バー コントロールでは、スクロール ボックスが指している現在の位置は、Value プロパティで示されます。

スクロール バーにおける最小位置は、水平スクロール バーでは左端, 垂直スクロール バーでは一番上です。また最大位置は、水平スクロール バーでは右端, 垂直スクロール バーでは一番下です。

スクロール バーは、主に次のようなプロパティを持っています。

プロパティ	説明
Min	スクロール バーの左端または一番上の値 (最小値)
Max	スクロール バーの右端または一番下の値 (最大値)
Value	スクロール バーが指している現在位置
LargeChange	ユーザーがスクロール バー上で、スクロール ボックスの上側 (左側) または下側 (右側) をクリックしたときの値の変化量
SmallChange	ユーザーがスクロール バーの端の矢印をクリックしたときの値の変化量

これらのプロパティは、すべてデザイン時にフォーム デザイナで設定できます。実行時には、(スクロール バーを無効に設定していない限り) ユーザーによるスクロール バーの操作に合わせて Value プロパティの値が変化します。それ以外の各プロパティは、コード内で変更しない限り値は変化しません。

実行時にコード内で Value プロパティに値をセットすると、スクロール ボックスの位置は自動的に更新されます。

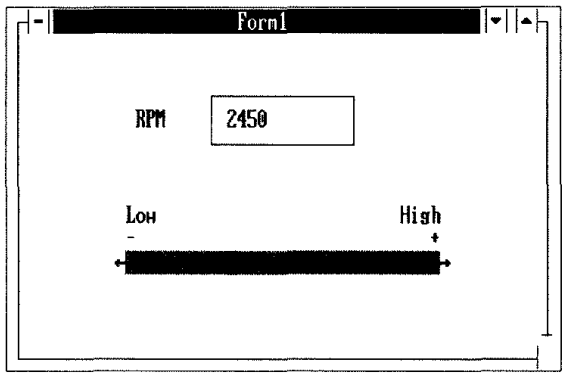
スクロール バーの値を変更するには、マウスによるクリックの他にもスクロール ボックスをドラッグする方法があります。移動後の値はスクロール ボックスの位置によって決定されますが、常に Min と Max の 2 つのプロパティの間の値を取ります。

スクロール バーの Value プロパティの利用

スクロール バーの Value プロパティには、スクロール バー内のスクロール ボックスの現在位置に対応した整数値がセットされています。

スクロール バー コントロールを使うと、相対的な位置は視覚的に分かりますが、正確な位置はユーザーには分かりません。正確な値を表示するための方法の 1 つは、スクロール バーの下にラベルを表示して、スクロール バーが変化するたびにその値を示すことです。ラベルの表示を更新するコードは、スクロール バーの Change イベント プロシージャの中に記述します。

次の例は、ラベル Label1 を使ってスクロール バー HScroll1 の位置を数値で表示します。スクロール バーの Change イベント プロシージャに、スクロール バーのカレントの値をラベルに表示するためのコードを記述します。




```
SUB Hscroll1_Change ()
    Label1.Caption = STR$(HScroll1.Value)
END SUB
```

4.11　実行時におけるコントロールの操作

コントロールの追加, 選択, 移動, サイズ変更, 削除, 表示, 非表示は、コード内でも設定できます。

実行時におけるコントロールの移動

実行時にコントロールを移動するには、次の 2 つの方法があります。

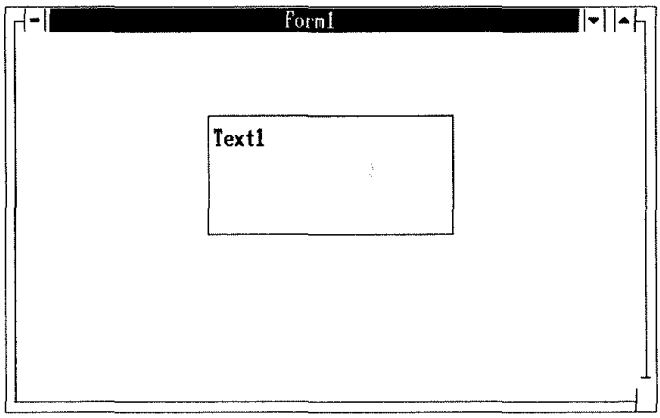
- 各コントロールの Top および Left プロパティを設定する。
- **MOVE** メソッドを使う。

Top および Left プロパティの設定

コントロールを移動する最も簡単な方法は、Left と Top プロパティに値を代入することです。

- Left プロパティは、そのコントロールの左上角と親フォーム (または親コントロール) の左端との距離を示します。
- Top プロパティは、そのコントロールの左上角と親フォーム (または親コントロール) の上端との距離を示します。

フォームは 0 を基準とする座標系を使っていて、親フォームの左上角が (0,0) になります。



コントロールを移動するには、そのオブジェクトの Left と Top の各プロパティを、次の代入文を使って設定します。

```
Text1.Left = Text1.Left + 10
Text1.Top = Text1.Top - 1
```

Top と Left の 2 つのプロパティを使ったコントロールの移動は非常に簡単ですが、Top と Left の各プロパティを別々に設定するため、コントロールははじめに水平、次に垂直の 2 段階で移動することになって動きがスムーズではありません。この動きをよりスムーズに行わせるには、**MOVE** メソッドを使います。**MOVE** メソッドは、スムーズな動きを実現するために、Top, Left, Width, Height の各プロパティを同時に変化させます。

MOVE メソッドの利用

MOVE メソッドの構文は、次のとおりです。

[*object* .] **MOVE** *left%* [, *top%* [, *width%* [, *height%*]]]

引数	説明
object	移動するフォームまたはコントロール。省略すると、カレントのフォームに対して適用されます (カレントのフォームが移動します)。

left%	新しく設定する Left プロパティの値を指定します。
top%	新しく設定する Top プロパティの値を指定します。
width%	新しく設定する Width プロパティの値を指定します。
height%	新しく設定する Height プロパティの値を指定します。

これらの引数のうち、必ず記述しなければならないのは *left%* だけです。それ以外の引数のいずれかを指定するときは、構文でその引数よりも前に来る引数を、すべて指定する必要があります。例えば *width%* を指定したいときは、同時に *left%* と *top%* も指定しなければなりません。指定しなかった引数の値は、変更されません。

次のステートメントでは、Text1 という名前のコントロールを移動しています。はじめのステートメントでは移動先の座標を、また次のステートメントでは現在位置に対する相対位置を、それぞれ指定しています。

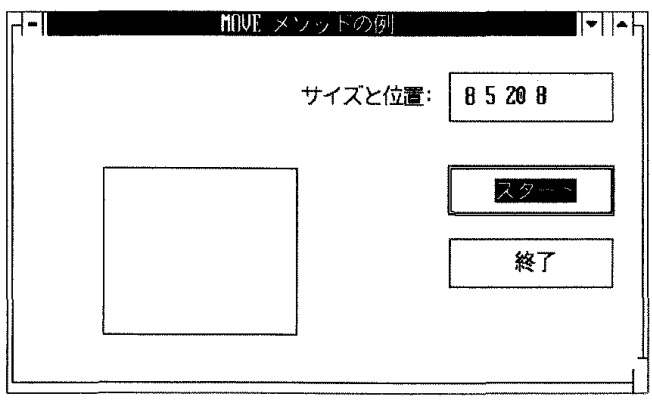
```
'テキスト ボックスを座標(10,20)に移動
Text1.MOVE 10, 20
'
'テキスト ボックスを現在位置から移動するとともに、その幅を変更
Text1.MOVE Text1.Left + 10, Text1.Top + 10, 8
```

注　意　Visual Basic の移動およびサイズ指定のステートメントでは、半角文字の 1 文字分を長さの単位として使っています。画面上で実現される物理的な長さは、ディスプレイのサイズによって異なります。

コントロールの移動のプログラム例

コントロールの移動は、さまざまなアプリケーションで利用できます。次に示すプログラムでは、**MOVE** メソッドと **RND** 関数を使って、コントロールの移動とサイズの変更を行います。

ユーザーが <スタート> ボタンをクリックすると、ランダムな引数が **MOVE** メソッドに引き渡され、ピクチャー ボックスの位置とサイズが変化します。これらの引数の値は、テキスト ボックスに表示されます。



タイマーを使うと、指定した時間間隔でピクチャー ボックスを移動できます。タイマーの値は、例えば 1000 ミリ秒 (1 秒) に設定します。

タイマー コントロールの使い方について詳しくは、第 13 章「マウス, キーボード, ドラッグ & ドロップ イベント」を参照してください。

▼ このアプリケーションを作成するには

1. フォーム デザイナでフォームを作成し、コントロールを追加します。
2. 各コントロールのプロパティを、次のように設定します。

オブジェクト	プロパティの設定	使用するイベント
フォーム	Caption = Move メソッドの例 WindowState = 2 (最大化)	
上のコマンド ボタン	CtlName = cmdStart Caption = スタート	Click
下のコマンドボタン	CtlName = cmdExit Caption = 終了	Click
ラベル	Caption = サイズと位置: Autosize = True (-1)	
テキスト ボックス	Text = (なし)	
タイマー	CtlName = Timer1 Enabled = True (-1) Interval = 1000 (1 秒)	Timer
ピクチャー ボックス	CtlName = Picture1	

参 考 フォームが最大化されているときは、[メニュー] キーを押すとメニュー バーとプロパティ バーが表示されます。

3. Form_Load イベント プロシージャで、タイマーの乱数系列を初期化します。

```
SUB Form_Load ( )
    RANDOMIZE          '乱数系列の初期化
END SUB
```

4. cmdStart_Click イベント プロシージャに、次に示すコードを記述します。

```
SUB cmdStart_Click()
    Timer1.Enabled = -1
END SUB
```

5. Timer1_Timer イベント プロシージャに、次に示すコードを記述します。

```
SUB Timer1_Timer()
    DIM LeftRight, UpDown, Wide, Tall AS INTEGER
                                        'MOVE メソッドの引数
    LeftRight = INT(45 * RND + 1)
    UpDown = INT(15 * RND + 1)
    Wide = INT(10 * RND + 1)
    Tall = INT(5 * RND + 1)
```

```
        Picture1.MOVE LeftRight, UpDown, Wide, Tall
                                    'ピクチャー ボックスの移動, サイズ変更
        Args$ = STR$(LeftRight) + STR$(UpDown) + STR$(Wide)
          ⇒ + STR$(Tall)
        Text1.Text = Args$ '引数をテキスト ボックスに表示
    END SUB
```

6. cmdExit_Click イベント プロシージャに、次に示すコードを記述します。

```
    SUB cmdExit_Click()
        SCREEN.HIDE
        END
    END SUB
```

このプログラムを実行すると、ピクチャー ボックスの移動とサイズ変更がランダムに行われます。<終了> ボタンを押すと、実行を終了します。

実行時におけるコントロールのサイズ変更

デザイン時ではなく実行時に、コントロールのサイズを変更する必要が生じる場合もあります。Visual Basic のアプリケーションでは、コントロールやフォームのサイズや形も、その位置と同様に実行時に変更できます。

先のプログラム例では、**MOVE** メソッドを使って実行時にピクチャー ボックス コントロールの移動とサイズ変更を行いましたが、この他にも次の 3 つのプロパティが、コントロールのサイズに影響します。

プロパティ	説明
Height	オブジェクトの高さを設定します。単位はキャラクタで 、境界線があるときはこれも含めた高さになります。
Width	オブジェクトの幅を設定します。単位はキャラクタで、境界線があるときはこれも含めた幅になります。
AutoSize	ラベルだけが持つプロパティで、 True (-1) に設定されていると、表示するテキストの長さに合わせて自動的にサイズを調節します。

注　意　Height と Width プロパティは、タイマーとメニューを除くすべてのフォームとコントロールが持っています。一方、AutoSize プロパティはラベルだけが持っています。

クリックするたびに大きくなるコマンド ボタン Command1 を作成するには、そのボタンの Click イベント プロシージャに次のコードを記述します。

```
SUB Command1_Click ()
    Command1.Height = Command1.Height + 1
    Command1.Width = Command1.Width + 2
END SUB
```

実行時におけるコントロールの表示/非表示の切り替え

オブジェクトを表示したり消したりする機能は、特にゲームやシミュレーションなどのアプリケーションで便利です。また、アプリケーションの途中で新しいオプションが表示されるようにしたいときも、この機能が有効です。この他にも、次のような場面でコントロールの表示/非表示の機能が役立ちます。

- 表示する前に、コントロールのプロパティを設定したいとき
- ダイアログ ボックスから必要のない機能を一時的に削除したいとき

例えば、状況によって必要になるオプション ボタンを作りたい場合、不要なときに無効 (EnabledをFalse) に設定する代わりに、いったん非表示にして、画面から消すことができます。

タイマー以外のすべてのコントロールは、True (-1) か False (0) のいずれかに設定可能な Visible プロパティを持っています。Visible プロパティを False に設定すると、コントロールはメモリ上にロードされたまま画面には表示されなくなります。

▼ **Text1 コントロールを非表示にするには**

- Visible プロパティを False (0) に設定します。

```
Text1.Visible = 0
```

Text1 はメモリ上にロードされたままなので、後で再表示するときには、非表示にしたときのプロパティの設定がそのまま有効です (ただし、非表示にしている間にプロパティを変更することもできます)。

▼ **Text1 コントロールを再び表示するには**

- Visible プロパティを True (-1) に設定します。

```
Text1.Visible = -1
```

また、1 行のステートメントで、コントロールの表示/非表示を交互に切り替えることも可能です。このステートメントでは、Visible プロパティの値 True (-1) とFalse (0) が論理演算子 NOT で交互に切り換えられることを利用しています。

```
SUB Command1_Click ()
    Text1.Visible = Not Text1.Visible
END SUB
```

ヒント NOT 演算子を使った True と False の切り替えは他のプロパティを操作するときにも使えます。False の値 (-1)は、2 進数で表わすとすべての桁が 1 になることに注意してください。

4.12　コントロール配列の利用

コントロール配列とは、同じ種類で同じ名前を持つコントロールの集合です。同じ CtlName を持つため、1 つのイベント プロシージャを共有することができます。CtlName 以外のプロパティはそれぞれ独自のものを持っているので、位置や大きさ，キャプションなどの外観は個別に設定できます。イベント プロシージャに、どのコントロールから呼び出されたかを伝えるために Index プロパティを使います。コードから配列中のコントロールにアクセスする場合は、Command1(0) のように CtlName の後の () の中で Index プロパティの値 (インデックス番号) を指定します。図 4.6 に、3 つのコマンド ボタンから構成されるコントロール配列を示します。

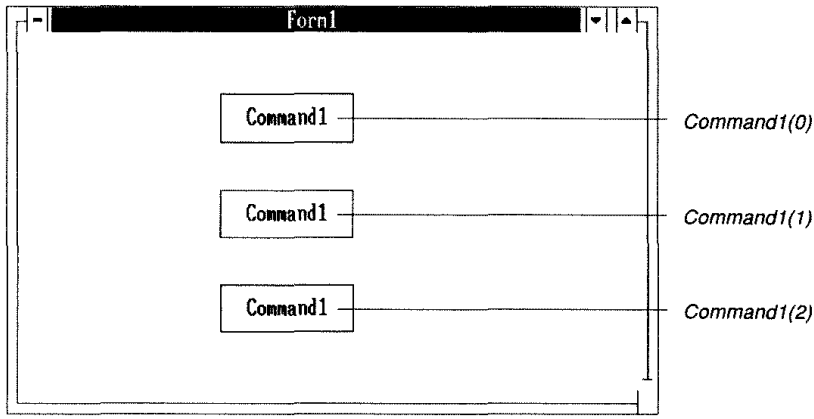


図 4.6　コントロール配列

コントロール配列は、複数のコントロールの間でイベント プロシージャを共有しようとするときに有効な手段です。例えば 3 つのオプション ボタンでコントロール配列を構成した場合は、どのオプション ボタンがクリックされてもまったく同じコードが実行されます。

参　考　コントロール配列は、メニュー コントロールに対してよく使われます。詳しくは、第 8 章「メニューの作成」を参照してください。

コントロール配列の要素は、最低 1 個から、最大で 254 個まで設定できます。

同じ配列に属するコントロールを識別するには、Index プロパティを使います。配列の中のコントロールがイベントを認識して、共通のイベント プロシージャを呼び出すときに、引数として Index プロパティの値が引き渡されます。

コントロール配列に対応する Command1_Click プロシージャの 1 行目は、次のようになります。

```
SUB Command1_Click (Index AS Integer)
```

Command1(0) がイベントを発生したときは引数 Index として 0 が引き渡され、Command1(1) がイベントを発生したときは 1 が引き渡されます。Command(0) と Command(1) は同一のイベント プロシージャ Command1_Click を呼び出します。

実行時にコントロールを生成するには、必ずコントロール配列の機能を使う必要があります。これは、新しくコントロールを作ってもそれに対応するイベント プロシージャが存在しないためです。コントロール配列を使うと、共通のイベント プロシージャを使うことができるので、アプリケーションの実行中に新しいコントロールを追加できます。例えば、フォーム内でデータの値を入力するテキスト ボックスを配列にしておくと、同じコードを共有するテキスト ボックスをアプリケーション実行中に追加することができます。

設計時におけるコントロール配列の作成

設計時にコントロール配列を作成する方法としては、次の 3 つがあります。

- コントロールの名前 (CtlName プロパティの値) を、既存の別のコントロールと同じ名前に変更する。
- すでに配置されているコントロールをコピーして、同じフォーム上に貼り付ける。
- Index プロパティに値を設定する。

▼ **コントロール名を変更してコントロール配列に加えるには**

1. フォーム デザイナで、コントロール配列にするコントロールを複数描きます (コントロールの種類は同じでなければなりません)。次に、どのコントロールを配列の先頭の要素にするか決めます。
2. 2 番目以降のコントロールの CtlName プロパティを、配列の先頭要素の CtlName と同じ名前に設定します。

   2 番目のコントロールの CtlName を書き換えるときに、コントロール配列を作成するかどうか確認するダイアログ ボックスが表示されるので、<Y/はい> を選択します。

例えば、コントロール配列の先頭要素の CtlName が cmdCtlArr であれば、配列に追加するコントロールに対しても、その CtlName を cmdCtlArr に設定します。これによって、コントロール配列を作成するかどうかを確認するメッセージが表示されます。ここで <Y/はい>を選ぶと、コントロール配列が作成されます。

この手順に従って配列に追加されたコントロールは、すべて同じ CtlName プロパティを持つことになります。また先頭要素の Index プロパティは 0 で、他のコントロールでは追加された順に 1 ,2 ,3...が自動的に Index プロパティに設定されます。それ以外のプロパティは最初に作成したときの設定のままです。CtlName 以外のプロパティは、(Index を含め) 個別に変更することができます。ただし、Index プロパティは配列内の他のコントロールと同じ値には設定できません。

▼ すでに配置されているコントロールをコピーしてコントロール配列に追加するには

1. フォーム デザイナで、コントロール配列の最初のコントロールを配置します。
2. [編集] メニューから [複写] コマンドを選びます。
3. フォームをクリックしてから、[編集] メニューの [貼り付け] コマンドを選びます。

   貼り付けを行うときに、コントロール配列を作成するかどうか確認するダイアログ ボックスが表示されるので、<Y/はい> を選択します。

   新しいコントロールの Index プロパティには自動的に、1 が割り当てられます。

配列に追加されたコントロールの Index プロパティの値は、そのコントロールが配列に追加された順に設定されます。この方法でコントロールを配列に追加していった場合、高さ, 幅, 色などの他のプロパティは、最初のコントロールと同じ設定になっています。ただし、実行時に配列に追加されるコントロールのプロパティを、フォーム デザイナで設定することはできません。

実行時におけるコントロール配列の作成

実行時に、既存のコントロール配列に対して、コントロールを追加したり削除したりできます。コントロール配列に要素を追加するときには **LOAD** ステートメントを、削除するときには **UNLOAD** ステートメントを、それぞれ次の構文に従って記述します。

LOAD *CtlName* (*Index%*)

UNLOAD *CtlName* (*Index%*)

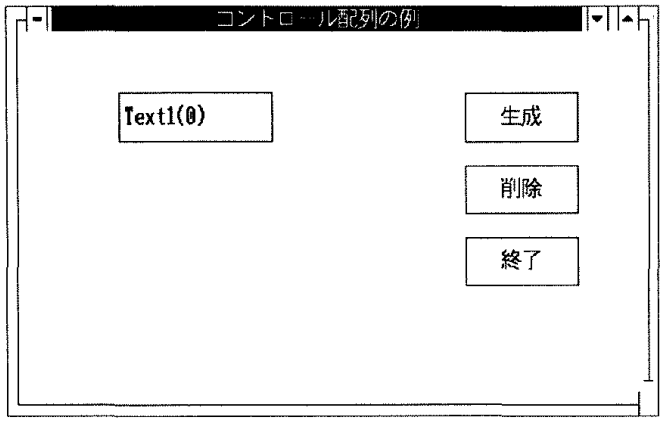
引数	説明
CtlName	コントロール配列に対して追加または削除するコントロールの名前
Index%	配列におけるそのコントロールの Index プロパティの値

既存のコントロール配列に対して新しいコントロールを追加すると、ほとんどのプロパティの設定は、配列の最初の要素と同じになります。ただし、Visible, Index, TabIndex の各プロパティの値は、新しい要素にコピーされません。

参　考　追加したコントロールが他のコントロールに重なってしまうのは好ましくありません。新しくコントロールを作成したら、Top や Left などの視覚的プロパティを設定し直してコントロールの位置を調整してください。

コントロール配列のプログラム例

次のプログラム例では、実行時にテキスト ボックスのコントロール配列に対してコントロールを追加したり削除したりします。<生成> ボタンをクリックすると 3 つのテキスト ボックスが新しく作成され、<削除> ボタンを押すと削除されます。また、<終了> ボタンをクリックするとプログラムが終了します。



▼ このアプリケーションを作成するには

1. フォーム デザイナで、3 つのコマンド ボタンと 1 つのテキスト ボックスを持つフォームを作成します。

2. 各コントロールのプロパティを次のように設定します。

オブジェクト	プロパティの設定	主なイベント
フォーム	Caption = コントロール配列の例	
テキスト ボックス	Text = Text1(0) CtlName = Text1 Index = 0	
上段のコマンド ボタン	CtlName = cmdCreate Caption = 生成	Click
中段のコマンド ボタン	CtlName = cmdRemove Caption = 削除	Click
下段のコマンドボタン	CtlName = cmdExit Caption = 終了	Click

作成したテキスト ボックスは、コントロール配列の最初の要素になります。

3. cmdCreate_Click イベント プロシージャに次のコードを記述することによって、実行時に 3 つのテキスト ボックスを追加し、これらのコントロールの Top, Text, Visible の各プロパティを設定します。

```
SUB cmdCreate_Click()
    DIM I AS INTEGER
    FOR I = 1 TO 3
        LOAD Text1(I)          '新しいテキスト ボックスの追加
        Text1(I).Top = Text1(I-1).Top + 3
                               '新しいテキスト ボックスの位置を調整
        Text1(I).Text = "Text1(" + FORMAT$(I) + ")"
                '新しいテキスト ボックスの名前とインデックス番号を表示
        Text1(I).Visible = -1          'テキスト ボックスを表示
    NEXT I
END SUB
```

<生成> ボタンがクリックされると、このイベント プロシージャによって新しいテキスト ボックスがコントロール配列に追加されます。これらのテキスト ボックスは縦に並べられ、新しい要素の名前と Index プロパティの値を表示します (Top プロパティを変更しなければ、これらのコントロールは同じ位置に作成され、重なってしまいます)。追加されたコントロールはデフォルトでは、Visible プロパティが False になっているため、画面に表示されません。Top やLeft などのプロパティによって位置を調整した後で Visible プロパティを True (-1) に設定します。

注　意　**LOAD** ステートメントで、Index プロパティの値として配列内ですでに使用されている値を指定すると、エラーが発生します。

---

4. cmdRemove_Click イベント プロシージャに、次のコードを記述します。

```
SUB cmdRemove_Click()
    DIM I AS INTEGER
    FOR I = 1 TO 3
    UNLOAD Text1 (I)
    NEXT I
END SUB
```

LOAD ステートメントで追加されたコントロールは、すべて **UNLOAD** ステートメントで削除できます。ただし、フォーム デザイナで設計時に作成されたコントロールについては、それがコントロール配列に含まれているかどうかにかかわらず、**UNLOAD** ステートメントでは削除できません。

4.13　タイマー コントロールの利用

タイマー コントロールは、ユーザーの操作に依存せずに、一定の時間間隔で発生する Timer イベントを認識します。主に、自動的に進行する処理のタイミングを決めるために使われます。各種のバックグラウンド プロセスを実行するときにも便利です。

タイマー コントロールは、イベントが発生する間隔を指定する Interval プロパティを持っていて、その値はミリ秒の単位で設定します。Enabled プロパティが False (0) に設定されていなければ、Timer イベントはほぼ一定の時間間隔で発生します。

タイマーを使ったプログラミングを行うときは、次に示す Interval プロパティの制約条件に注意してください。

- Interval プロパティの設定可能範囲は 0 ～ 64,767 ミリ秒であり、最長でも 1 分間を少し上回る程度 (約 64.8 秒) です。
- 精度の点で、設定した時間間隔どおりに駆動するという保証はありません。精度を高めるには、タイマー内部で計時するだけでなく、システム クロックをチェックする必要があります。
- 実際に計測できる時間の最小単位は、ハードウェアに依存します。

タイマー コントロールは常に、フォームと結び付けられています。このため、タイマーを使ったアプリケーションを作成するときは、少なくとも 1 つはフォームを作成する必要があります。ただしフォームは、非表示に設定してロードすることもできます。

注 意 Visual Basic では、従来からの方法である、ON TIMER ステートメントを使ったタイマー割り込みもサポートしています。ただし、フォームが表示されているときには利用できません。

フォームへのタイマー コントロールの配置

フォーム上にタイマー コントロールを配置する方法は、他のコントロールと同じです。ただしタイマーコントロールは表示できないので、Visible, Height, Width などのプロパティを持ちません。Left および Top プロパティはありますが、デザイン時に他のコントロールと重ならないようにするためで、実行時には意味を持ちません。

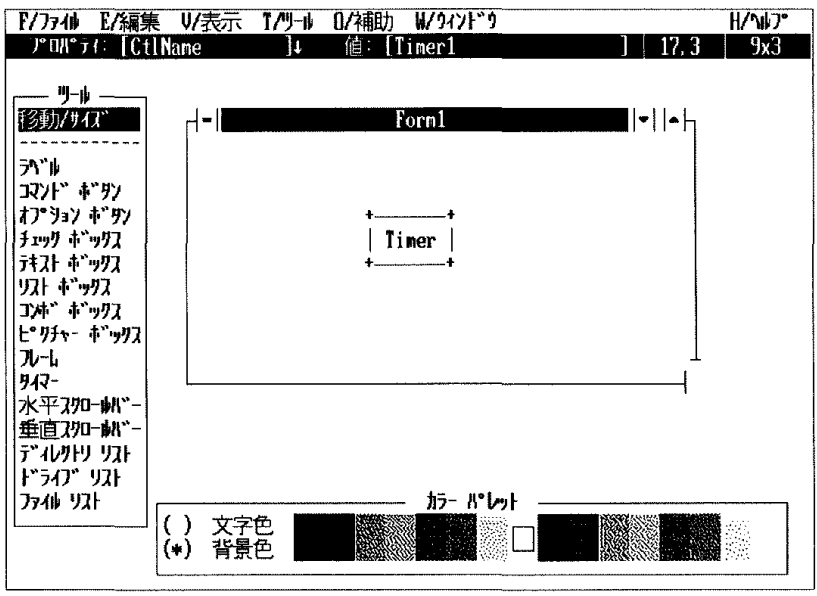


図 4.7 タイマー コントロール

タイマー コントロールの初期化

タイマー コントロールでは、主に次の 4 つのプロパティを使います。

プロパティ	設定
CtlName	タイマーのコントロール名を示します。
Enabled	フォームのロードと同時にタイマーのカウントを開始したいときはこのプロパティを True (-1) に、そうでないときは False (0) に、それぞれ設定します。False に設定したときは、外部のイベント (例えばコマンド ボタンのクリックなど) によってタイマーの計測が開始されるようにプログラミングしておきます。
Interval	タイマーの時間間隔の設定値 (単位はミリ秒)。
Index	タイマーがコントロール配列の要素であるときに使います。

タイマー コントロールと他のオブジェクトでは、Enabled プロパティの持つ意味が異なる点に注意してください。タイマー以外のオブジェクトでは、Enabled プロパティはイベントを認識するかどうかだけを示すのに対し、タイマー コントロールでは、時間の計測を行うかどうか自体を示します。

注　意　タイマー イベントの頻度が増すと、それだけ多くの CPU 時間がタイマー イベントの処理に割かれるため、アプリケーション全体の実行速度は低下します。したがって、必要のないときは特に時間間隔の短いタイマーを設定しないようにしてください。

タイマー イベントへの応答

タイマー コントロールは、設定時間が経過すると発生するTimer イベントを認識します。

タイマー コントロールを利用したアプリケーションのうち、非常に簡単でしかもよく使われるのがデジタル クロックです。デジタル クロックのメカニズムを理解すると、アラーム クロック, ストップ ウォッチなどの各種の時計も作れるようになります。

次に示すデジタル クロックのアプリケーションでは、境界線付きのラベルとタイマーを使っています。

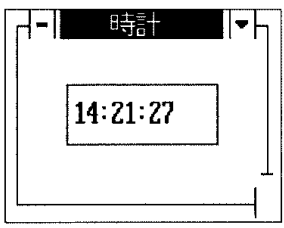


図 4.8　デジタル クロックのアプリケーション例

実行時には、タイマー コントロールは表示されません。

ラベルの BorderStyle プロパティは 1 (細線)に設定し、Caption プロパティはデフォルトで設定される Label1 を削除し、何も表示されないようにします。また、タイマーの Interval プロパティは 1000, Enabled プロパティは True (-1) に、それぞれ設定します。

このアプリケーションの唯一のプロシージャは、次に示す Timer のイベント プロシージャです。

```
SUB Timer1_Timer ()
    Label1.Caption = TIME$
END SUB
```

Timer イベント プロシージャは、システム時間を表示するために**TIME$** 関数を呼び出します。この関数は現在の時刻を hh:mm:ss の形式の文字列で返します。

アラーム クロックの作成

デジタル クロックの一部を修正して、簡単にアラーム クロックを作成することができます。次に示すアラーム クロックのアプリケーションは、文字列変数 AlarmTime にアラーム時刻を保持し、クロックが進むたびにアラーム時刻になったかどうかをチェックします。時刻が来るとユーザーに知らせます。

アラーム時刻を格納する文字列変数は、フォームのモジュールレベルで宣言します。

```
DIM SHARED AlarmTime AS STRING
```

アラーム時刻を設定するには、フォームの空白部分をクリックするようにします (メニュー コマンドやコマンド ボタンにこの機能を割り当てることもできます)。このプロシージャでは、ユーザーが時刻を入力するのに **INPUTBOX$** 関数を使っています。新しい時刻が入力されて <了解> ボタンが押されると、その時刻が変数に代入されます。ユーザーが <取消> ボタンをクリックすると、**INPUTBOX$** 関数は null 文字列 ("") を返します。

```
SUB Form_Click ()
    Temp$ = INPUTBOX$("新しい時刻を入力してください:")
    IF Temp$ <> "" THEN AlarmTime = Temp$
END SUB
```

Timer イベント プロシージャでは、時刻表示の更新を行った後に現在時刻と AlarmTime 変数に保持されている文字列との比較を行い、等しければビープ音でユーザーに知らせます。

```
SUB Timer1_Timer ()
    Label1.Caption = TIME$
    IF TIME$ = AlarmTime THEN
        BEEP
        MSGBOX "時間です！"
    END IF
END SUB
```

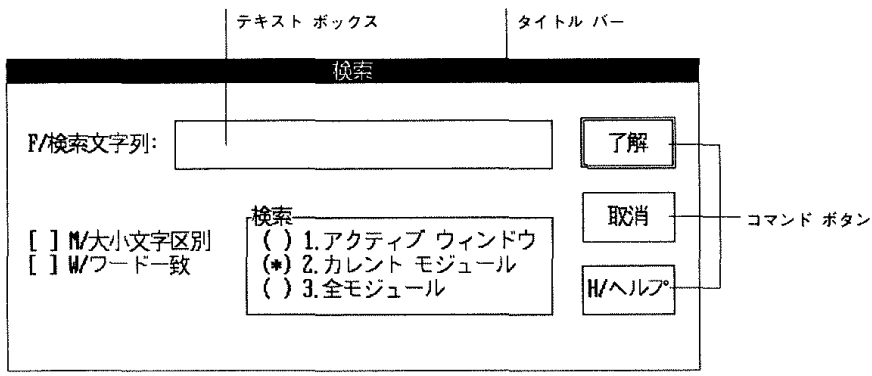
アラーム時刻に達した時点でアプリケーションがアクティブでないときは、アラーム音は発生しません。次のように Timer イベント プロシージャを修正すると、アプリケーションがアクティブになったときに設定時間を過ぎていれば、アラーム音を鳴らすようにできます。

```
SUB Timer1_Timer ()
    CONST TRUE = -1, FALSE = 0
    STATIC AlarmSounded AS INTEGER
    Label1.Caption = TIME$
    IF AlarmTime = "" THEN EXIT SUB
    If TIME$ >= AlarmTime and Not AlarmSounded THEN
        BEEP
        MSGBOX "時間です！"
        AlarmSounded = TRUE
    ELSEIF TIME$ < AlarmTime THEN
        AlarmSounded = FALSE
    END IF
END SUB
```

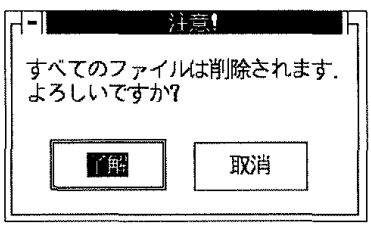
5 ダイアログ ボックスの作成

“ダイアログ ボックス” は、ユーザーが入力や確認の操作を行うために一時的に表示されるウィンドウです。Visual Basic のプログラミング環境でよく利用されるダイアログ ボックスには、デバッグ メニューの [ウォッチ追加], 実行メニューの [EXE ファイル作成], 表示メニューの [フォーム選択] の各コマンドを選んだときに表示されるものなどがあります。

例えば検索機能で使われる [検索] ダイアログ ボックスには、検索する文字列を入力するためのテキスト ボックスが表示されます。



ダイアログ ボックスは、エラー メッセージをはじめとする各種のメッセージを表示するときにも使われます。



ダイアログ ボックスは、ユーザーとの対話のために一時的に表示されるウィンドウの総称であり、ダイアログ ボックスであるための厳密な定義があるわけではありません。移動可能かどうかやモーダル ウィンドウであるかどうか、どんなコントロールが含まれているかなど、さまざまなダイアログ ボックスがあります。ただし通常は、ダイアログ ボックスにはメニュー バー，水平および垂直スクロール バー，最小化および最大化ボタン，ステータス バーなどはありません。

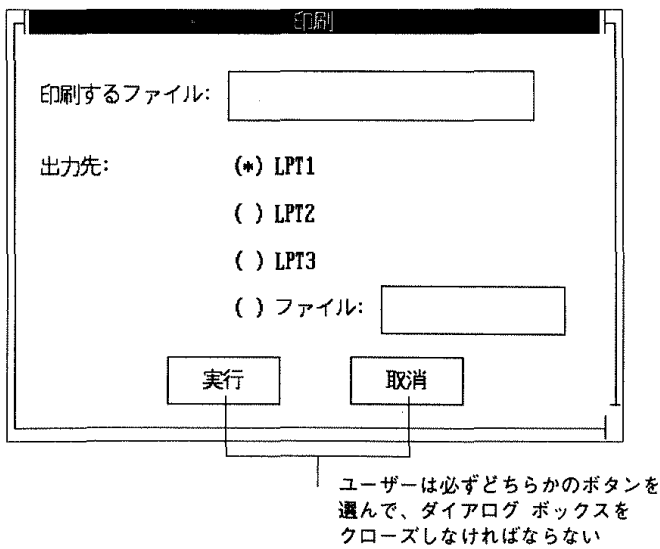
ダイアログ ボックスを作成するには、次の 2 つの方法があります。

- **INPUTBOX$** 関数，**MSGBOX** ステートメント，**MSGBOX** 関数のいずれかを使って、定義済みの (Visual Basic が標準で用意している) ダイアログ ボックスを表示する。
- フォームを作成して、ダイアログ ボックスとして表示する。

定義済みのダイアログ ボックスは, コードを 1 行記述するだけで簡単に利用できますが、あらかじめ決められているフォーマットしか使えません。より複雑なダイアログ ボックスを表示したいときは、フォーム上にさまざまなコントロールを配置して、独自のダイアログ ボックスを作成してください。

5.1　モーダル ダイアログ ボックスと モードレス ダイアログ ボックス

ダイアログ ボックスには、モーダル ダイアログ ボックスとモードレス ダイアログ ボックスの 2 種類があります。“モーダル” ダイアログ ボックスは、ユーザーがなんらかの情報 (例えばファイル名) を入力して <了解> や <実行> などのボタンを押すか、<取消> ボタンを押してダイアログをキャンセルするまで、別のフォームやダイアログ ボックスに移ることができないダイアログ ボックスです。つまり、そのダイアログ ボックスがクローズされるまで、フォーカスを保持します。



通常のダイアログ ボックスは、モーダル ダイアログ ボックスとして作られます。つまり、<了解> や <実行>，<取消> などのボタンをクリックしてダイアログ ボックスを閉じなければ、次の処理に進むことができないようにしておきます。

“モードレス” ダイアログ ボックスは、そのダイアログ ボックスを開いたまま、別のフォームやダイアログ ボックスにフォーカスを移すことができるダイアログ ボックスです。つまり、ダイアログ ボックスが表示されていても、アプリケーションの別の場所では作業を続けることができます。

5.2 定義済みのダイアログ ボックスの利用

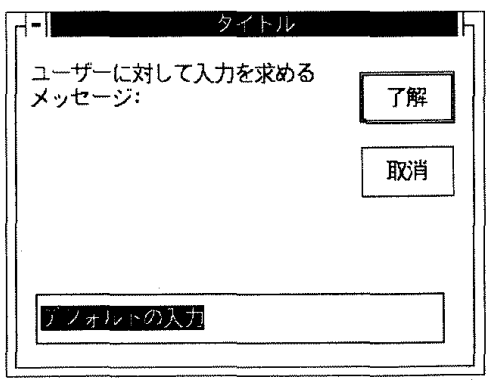
最も簡単にアプリケーションにダイアログ ボックスを追加する方法は、定義済みのダイアログ ボックスを使うことです。

次の表は、Visual Basic のアプリケーションに定義済みのダイアログ ボックスを追加する方法を示しています。

方法	利用目的
INPUTBOX$ 関数	入力を求めるメッセージを含むダイアログ ボックスを表示して、ユーザーが入力したテキストを返します。
MSGBOX ステートメント	メッセージを含むダイアログ ボックスを表示します。
MSGBOX 関数	メッセージを含むダイアログ ボックスを表示して、ユーザーが選んだコマンド ボタンを示す値を返します。
CMNDLGF.FRM と CMNDLG.BAS	アプリケーションでよく使われる、[オープン]，[保管]，[印刷]，[検索]，[置換]，[カラー パレット]，[...について] の各ダイアログ ボックスを表示します。

INPUTBOX$ による入力の取得

ユーザーからテキスト データの入力を受け取るには、**INPUTBOX$** 関数を使います。**INPUTBOX$** 関数は、文字列の入力を求めるダイアログ ボックスを表示します。



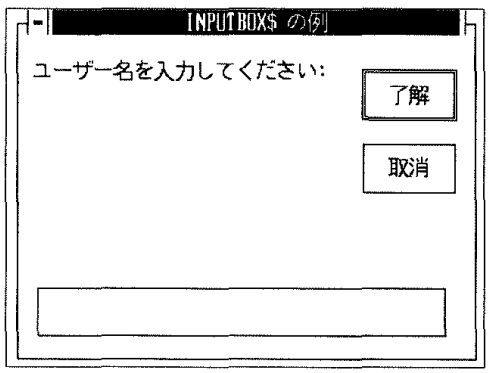
INPUTBOX$ 関数で表示されるダイアログ ボックスは、ほとんど変更できない点に注意してください。変更できるのは、タイトル，ユーザーに対して表示するメッセージ，デフォルトの入力，ダイアログ ボックスの画面上の位置だけです。

INPUTBOX$ 関数の構文は次のとおりです。

**INPUTBOX$** (*prompt$* [, *title$* [, *default$* [, *xpos%*, *ypos%*]]])

引数	内容
prompt$	ユーザーに対して表示されるメッセージ
title$	タイトル バーに表示されるテキスト
default$	テキスト ボックスに表示されるデフォルトの入力
xpos%	画面の左端からの距離
ypos%	画面の上端からの距離

例えば、ユーザー名の入力を求める次のようなダイアログ ボックスを考えてみます。



このダイアログ ボックスを表示するには、**INPUTBOX$** 関数を使って次のように指定します。

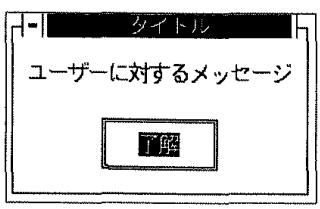
```
User$ = INPUTBOX$("ユーザー名を入力してください:", "INPUTBOX$ の例")
```

INPUTBOX$ 関数は、ユーザーが <了解> ボタンを選ぶと入力された文字列を返し、<取消> ボタンを選ぶと null 文字列 ("") を返します。

INPUTBOX$ 関数で表示されるのはモーダル ダイアログ ボックスです。つまりユーザーは、ダイアログ ボックスをクローズしない限り、アプリケーション中の別のフォームにフォーカスを移せません。

MSGBOX によるメッセージの表示

エラー，警告，注意などの短いメッセージを表示するときは、**MSGBOX** を使います。ユーザーは、メッセージを読み終えたら <了解> を選び、ダイアログ ボックスをクローズします。



MSGBOX には関数とステートメントの両方があり、それぞれ似た機能を持っています。**MSGBOX** 関数が、ユーザーが選んだボタンを示す値を返すのに対し、ステートメントは値を返しません。

MSGBOX 関数の構文は次のとおりです。これは関数であり、式の中で使います。

MSGBOX (*msg$* [, *type%* [, *title$*]])

MSGBOX ステートメントの構文は次のとおりです。引数をカッコで囲まないことに注意してください。**MSGBOX** ステートメントは値を返しません。

MSGBOX *msg$* [, *type%* [, *title$*]]

引数	意味
msg$	表示するメッセージ (最大で 1024 バイト)
type%	表示するボタンの組み合わせを示す整数
title$	タイトル バーに表示される文字列

引数 *type%* は表示するコマンド ボタンの組み合わせを指定する数値です。

値	表示されるボタンの組み合わせ
0 (デフォルト)	<了解>
1	<了解>, <取消>
2	<A/中止>, <R/再試行>, <I/無視>
3	<Y/はい>, <N/いいえ>, <取消>
4	<Y/はい>, <N/いいえ>
5	<R/再試行>, <取消>

この値に、次の表の値を加えて指定すると、何番目のボタンをデフォルトのボタンにするかを設定できます。また、<取消> ボタンを含む組み合わせでは、<取消> ボタンがキャンセル ボタンになり、ユーザーが [ESC] キーを押したときに選択されます。

値	デフォルトのボタン
0	1 番目
256	2 番目
512	3 番目

例えば *type%* に 514 を指定すると、<A/中止>, <R/再試行>, <I/無視> の 3 つのボタンが表示され (値 2)、3 番目のボタンである <I/無視> がデフォルト ボタンになります (値 512)。

MSGBOX 関数は、押されたボタンを示す整数値を返します。次のプログラム例では、メッセージと、<了解>, <取消> の 2 つのボタンが表示されます。

```
msg$ = "すべてのファイルは削除されます。よろしいですか？"
```

```
OkCancel% = 1              '<了解>, <取消> の 2 つのボタンを表示
Default% = 256             '<取消> ボタンがデフォルト
Response% = MSGBOX (msg$, OkCancel% + Default%, "注意!")
```

MSGBOX 関数の戻り値は次のようになります。

戻り値	対応するボタン
1	<了解>
2	<取消>
3	<A/中止>
4	<R/再試行>
5	<I/無視>
6	<Y/はい>
7	<N/いいえ>

上のプログラム例では、ユーザーが <了解> を選ぶと Response% の値は 1、<取消> を選ぶと 2 になります。

コモン ダイアログ ツールキットの利用

Visual Basic では、さまざまなアプリケーションで共通に利用できるダイアログ ボックスを集めた、コモン ダイアログ ツールキットを用意しています。次の 7 種類のダイアログ ボックスを表示するプロシージャが含まれています。

- オープン (FileOpen)
- 保管 (FileSave)
- 印刷 (FilePrint)
- 検索 (FindText)
- 置換 (ChangeText)
- カラー パレット (ColorPalette)
- このプログラムついて (About)

▼ コモン ダイアログ ボックスを利用するには

1. インクルード ファイル CMNDLG.BI が、カレント ディレクトリにあることを確認します。
2. [ﾌｧｲﾙ] メニューから [ファイル追加] コマンドを選び、コード モジュール CMNDLG.BAS をプロジェクトに加えます。このモジュールの中には、コモン ダイアログ ボックスを表示するためのプロシージャが含まれています。
3. 同様の手順で、フォーム モジュール CMNDLGF.FRM を追加します。
4. CMNDLG.BAS に含まれるプロシージャを呼び出すコードを、アプリケーションに加えます。

コモン ダイアログ ボックスの使用例として、CMNDLG.BAS のモジュールレベルに、各プロシージャの呼び出しを行うサンプル コードが含まれています。このサンプルを実行するには、[実行] メニューの [実行開始ファイル設定] コマンドを選び、CMNDLG.BAS を実行開始ファイルに設定してください。実行すると、最初のダイアログ ボックスが表示されます。<取消> ボタンを押すか [ESC] キーを押すと、他のダイアログ ボックスが順に表示されます。

コモン ダイアログ ツールキットに含まれるプロシージャを呼び出すには、それぞれ次のようにします。

FileOpen *filename$, pathname$, pattern$, title$, fcolor$, bcolor$, flags%, cancel%*

FileSave *filename$, pathname$, pattern$, title$, fcolor$, bcolor$, flags%, cancel%*

FilePrint *copies%, fcolor$, bcolor$, cancel%*

FindText *findtext$, fcolor$, bcolor$, options%, flags%, cancel%*

ChangeText *findtext$, changetext$, fcolor$, bcolor$, options%, flags%, cancel%*

ColorPalette *colornum%, fcolor$, bcolor$, cancel%*

About *aboutmsg$, fcolor$, bcolor$, flags$*

ユーザーが <取消> を選ぶと、*cancel%* 引数で指定した変数に真 (-1) が格納されます。それぞれのプロシージャの詳細については、CMNDLG.BAS 中のコメントを参照してください。

次のプログラム例では、¥VBDOS ディレクトリ中の拡張子 .TXT を持つファイルの一覧を表示し、ファイルのオープンを行う、[オープン] ダイアログ ボックスを表示します。

```
PName$ = "VBDOS"
FileOpen FName$, PName$, "*.TXT", "オープン",0 ,7 ,0 ,Cancel%
IF NOT Cancel% THEN
    OPEN PName$ + "¥" + FName$ FOR INPUT AS #1
END IF
```

コモン ダイアログ ツールキットすべてを組み込むとかなりのサイズになり、実行速度も低下するため、通常は CMNDLG.BAS と CMNDLGF.FRM を別の名前でコピーし、必要なルーチンだけを残してから、アプリケーションに組み込むようにします。この場合、CMNDLG.BI を参考にして、必要な宣言をモジュール レベルに記述してください。

5.3 独自のダイアログ ボックスの作成

フォーム上にコントロールを配置して、アプリケーション独自のダイアログ ボックスを作成することができます。通常は、コマンド ボタン，オプション ボタン，テキスト ボックスなどのコントロールを使います。このダイアログ ボックスのロードや表示を行ったり、アクティブにするためには、それぞれ対応する Basic コードを記述する必要があります。

参　考　ファイル，ディレクトリ，ドライブの各リスト ボックスを含むダイアログ ボックスの作成方法については、第 10 章「ファイル システム コントロールの利用」を参照してください。

フォームの作成とカスタマイズ

ダイアログ ボックスを作成するには、まずフォーム デザイナで新しいフォームを生成し、アプリケーション内でのフォームのサイズと位置を決めます。例えば、フォーム デザイナの画面上でフォームを中心に配置すると、アプリケーションの実行時にも画面の中心に表示されます。

新規にダイアログ ボックスを作成する代わりに、既存のダイアログ ボックスに変更を加えて利用することもできます。

▼ **既存のダイアログ ボックスをカスタマイズするには**

1. プロジェクトにフォームを追加します。
2. [ﾌｧｲﾙ] メニューから [新規保管] コマンドを選び、新しいファイル名を入力します。
3. フォーム デザイナで、フォームの外観をカスタマイズします。
4. プログラミング環境で、イベント プロシージャをカスタマイズします。

汎用的なダイアログ ボックスを作成しておけば、多くのアプリケーションで利用できます。

タイトルの変更

ダイアログ ボックスのタイトルは、一見してその機能が分かるようなものにします。

▼ **ダイアログ ボックスのタイトルを変更するには**

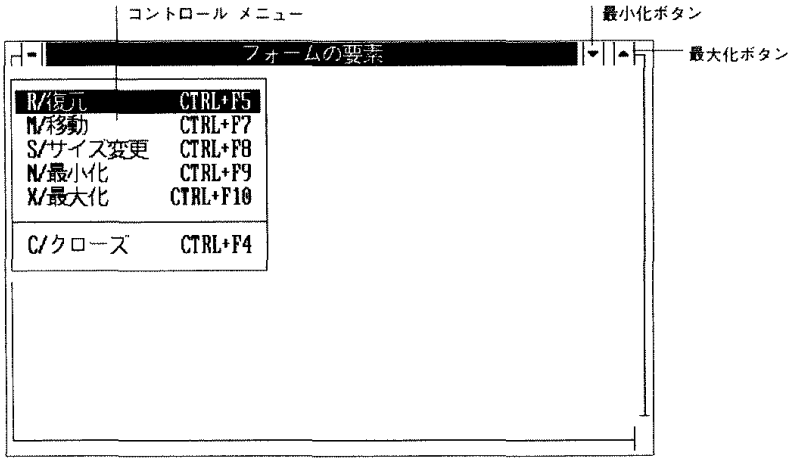
- フォーム デザイナでフォームの Caption プロパティを変更します。ここで設定したテキストはタイトル バーに表示されます。

コード中からフォームの Caption プロパティを設定することもできます。例えば次のようにします。

```
frmFind.Caption = "検索"
```

コントロール メニュー，最大化および最小化ボタンの削除

一般に、ダイアログ ボックスは必要なときだけ一時的に表示されるもので、ダイアログ ボックスが閉じられるまでは他の操作ができないよう (モーダル) に設定されます。このため通常は、サイズ変更，最小化，最大化などの操作は不要です。新しいフォームは、コントロール メニュー ボックス，最大化および最小化ボタンが付いた状態で生成されますが、ダイアログ ボックスとして使うときには、不要な要素を削除してください。



例えば、サイズ変更や最小化ができないダイアログ ボックスを作成するには、次のようにプロパティを設定します。

削除する機能	プロパティの設定方法
サイズ変更	BorderStyle = 1 - 細線 (サイズ固定), 3 - 太線 (サイズ固定), 5 - 極太 (サイズ固定) のいずれか
移動	BorderStyle = 0 - なし
最大化	MaxButton = False
最小化	MinButton = False
コントロール メニュー ボックス	ControlBox = False

注 意 MaxButton と MinButton を False (0) に設定すると、コントロール メニュー ボックスの [復元], [最小化], [最大化] の各コマンドが無効になります。また、BorderStyle を 1, 3, 5 のいずれかに設定すると、コントロール メニュー ボックスの [サイズ変更] コマンドが無効になります。

コントロール メニュー ボックスを削除する場合は、ダイアログ ボックスを終了するための方法を別に用意する必要がある点に注意してください。通常は、ダイアログ ボックスに <了解>, <取消>, <終了> などのコマンド ボタンを用意します。

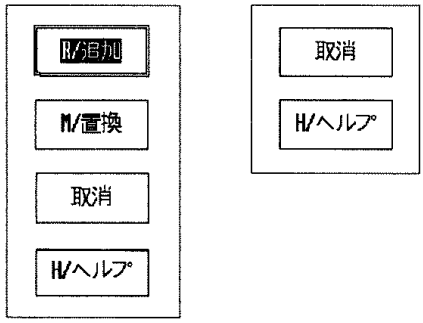
コントロールの追加

フォームにコントロールを追加するには、フォーム デザイナのツールボックスを使います。ツールボックスを使ったコントロールの作成方法や各コントロールの特性については、第 3 章「イベントドリブン アプリケーションの作成」と第 4 章「コントロールの作成と利用」を参照してください。

ダイアログ ボックスへのコマンド ボタンの追加

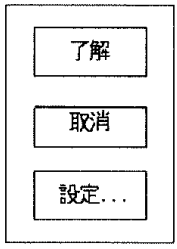
モーダル ダイアログ ボックスには、ダイアログ ボックスを終了させるためのコマンド ボタンが少なくとも 1 個は必要です。通常は、操作を実行するためのコマンド ボタンと、変更を行わずにダイアログ ボックスを終了するためのコマンド ボタンの 2 つを用

意します。最も多く使われるのは <了解> と <取消> ボタンですが、他のボタンの組み合わせを使ってもかまいません。



コマンド ボタンは、ダイアログ ボックスの下の方または右側に配置し、1 番左または上のボタンをデフォルト ボタンとして設定するのが一般的です。

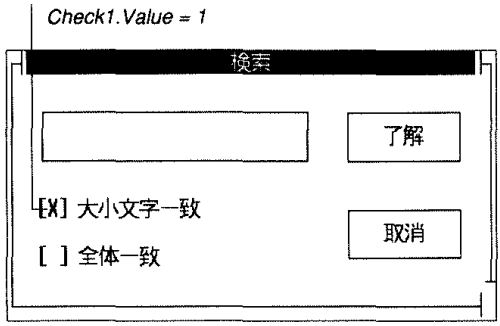
コマンド ボタンがクリックされたら、さらに別のダイアログ ボックスを表示することも可能です。通常は、コマンド ボタンに表示されるテキスト (Caption プロパティの値) の最後に ... を付けると、別のダイアログ ボックスが表示されることを示します。



メッセージを表示するダイアログ ボックスは、普通はコマンド ボタンだけを持ちます。例えば、よく見られるダイアログ ボックスに、プロンプトのメッセージと <了解> と <取消> の 2 つのボタンだけを持つものがあります。この場合、<了解> が押されればある動作を行い、<取消> が押されれば動作は行わないようにします。

コントロールのデフォルトの設定

ダイアログ ボックスにコントロールを追加するときは、各タイプのコントロールに対して、それぞれデフォルトの状態を指定できます。例えばチェック ボックスでは、Value プロパティを 1 (チェック) に設定すると、そのチェック ボックスはチェックされた (オン) 状態で表示されます。



ダイアログ ボックス内に複数のコマンド ボタンがある場合、デフォルト ボタンを設定することできます。

▼ ダイアログ ボックスのデフォルトのコマンド ボタンを指定するには

- そのコマンド ボタンの Default プロパティを True (-1) に設定します。

Default プロパティを True に設定できるのは、1 つのフォームに対してコマンド ボタン 1 つだけです。ユーザーが [ENTER] キーを押すと、このコマンド ボタンに対して Click イベントが発生します。この機能は、テキスト ボックスのような コントロールを使う場合に便利です。つまり、ユーザーがテキスト ボックスにデータを入力した後で、<了解> ボタンを選ぶ代わりに [ENTER] キーを押すと、Click イベントが発生するようにできます。

▼ フォームが表示されたときにコマンド ボタンにフォーカスを設定するには

- あるコマンド ボタンの TabIndex プロパティを 0、TabStop プロパティを True に、それぞれ設定します。

フォームが表示された直後には、TabIndex が最小のコントロールがフォーカスを持っています。[TAB] キーやマウスを使って別のコントロールにフォーカスを移動しない限り、ユーザーが [ENTER] キーを押すと、このコマンド ボタンに対する Click イベントが発生します。

▼ ダイアログ ボックスの <取消> ボタンの動作を指定するには

- あるコマンド ボタンの Cancel プロパティを True (0) に設定します。

Cancel プロパティを True に設定できるのは、1 つのフォームに対してコマンド ボタン 1 つだけです。ユーザーが [ESC] キーを押すと、このコマンド ボタンに対して Click イベントが発生します。通常は、<取消> ボタンをデフォルトのコマンド ボタンにします。

ヒント 一般的には、選ばれる確率が最も高い操作か、最も安全な操作をデフォルトの操作に設定します。例えばテキストの置換を行うダイアログ ボックスでは、<すべて置き換え> のような機能をデフォルトのボタンに設定すべきではありません。

オプションの無効化

ユーザーに触れさせたくない選択肢は、無効 (disable) に設定することができます。無効に設定されたコントロールは、画面ではデフォルトで薄い色で表示されます。

例えば、入力されたパスワードによって、データを削除できる人かを判断して、削除に関する機能を無効にしたい場合などに使います。

▼ **コントロールを選択できないようにするには**

- コントロールの Enabled プロパティを False (0) に設定します。

複数のフォームの表示と消去

フォームを使って独自のダイアログ ボックスを作成した場合、アプリケーションには複数のフォームが含まれることになります。ダイアログ ボックス用のフォームも、ほかのフォームと同様に取り扱われます。実行開始モジュールに記録されているフォームは、アプリケーションの実行開始時に自動的にロード，表示されますが、他のフォームに対しては、メモリへのロードと表示を行うコードを記述する必要があります。

また、ダイアログ ボックスを画面から消去するために、フォームをメモリ上からクリアするか、非表示にするコードを追加する必要があります。

フォームを表示/非表示にする方法を、次に示します。

目的の操作	コード内での記述
フォームをメモリ上にロードするだけで、画面には表示しない	**LOAD** ステートメントか、フォームのプロパティへの参照を使う (暗黙のロード)
フォームをロードして表示する	**SHOW** メソッドを使う
ロードされているフォームを表示する	Visible プロパティを True (-1) に設定するか、**SHOW** メソッドを使う
フォームを非表示にする	Visible プロパティを False (0) に設定するか、**HIDE** メソッドを使う
フォームをアンロードする	**UNLOAD** ステートメントを使う

例として、frmNew という名前のフォームで構成されるダイアログ ボックスを考えてみます。ユーザーが mnuCmd という名前のメニュー コマンドを選んだときにモーダル ダイアログ ボックス frmNew が表示されるようにするには、メニュー コマンドの Click イベントに次のコードを追加します。

```
SUB mnuCmd_Click ()
    frmNew.SHOW 1              'モーダル ダイアログ ボックスの表示
END SUB
```

SHOW メソッドはフォームをロードして、Visible プロパティを True (-1) に設定します。**SHOW** メソッドの引数はダイアログ ボックスのスタイルを示しており、0 (デフォルト) はモードレス、1 はモーダルのダイアログ ボックスを意味します。

ユーザーが <了解> または <取消> を選んだときにダイアログ ボックスを終了するには、**UNLOAD** ステートメントか **HIDE** メソッドを使います。

```
UNLOAD frmNew
```

または

```
frmNew.HIDE
```

UNLOAD ステートメントはダイアログ ボックスをメモリから消去します。このため、コントロールのプロパティやフォーム モジュール内の変数などは保存されません。一方、**HIDE** メソッドは Visible プロパティを False (0) に設定することによって非表示にします。**HIDE** メソッドで非表示の状態になっているフォームが **SHOW** メソッドによって再び表示されたとき、コントロールのプロパティや変数などはすべて保存されています。ただし、フォームやピクチャー ボックス上のテキストはクリアされるので、必要に応じて AutoRedraw プロパティの設定を行ってください。

ヒント　メモリの容量が限られているときは、フォームを非表示にするよりもアンロードするほうがよいでしょう。アンロードすると、その分だけメモリが解放されます。一方、ダイアログ ボックスが頻繁に使われるときは、アンロードせずに非表示にすることも意味があります。フォームを非表示にしても、プロパティの設定値、プリンタ出力、動的に生成されたコントロールなどのフォームに付加されたデータは保存されるため、非表示にしたフォームのプロパティやコントロールをコード中から参照することもできます。

$FORM メタコマンド

1 つのアプリケーションが複数のモジュールから構成されているとき、別のフォーム モジュールのフォームを参照するには、**$FORM** メタコマンドで宣言しておく必要があります。プログラミング環境で作業していれば、プログラミング環境がモジュール間の参照関係を判断して、自動的に **$FORM** メタコマンドを追加します。別のエディタを使ってコードを記述する場合は、これらをプログラマ自身が追加する必要があります。

例えば、アプリケーション全体から利用したいダイアログ ボックスがあるときは、ダイアログ ボックスを参照しているすべてのモジュールのモジュールレベルに、**$FORM** メタコマンドを挿入します。

```
'モジュールレベル コード
'$FORM Dialog1
```

$FORM メタコマンドでは、フォームのファイル名ではなく、FormName プロパティの値を指定しなければならないことに注意してください。

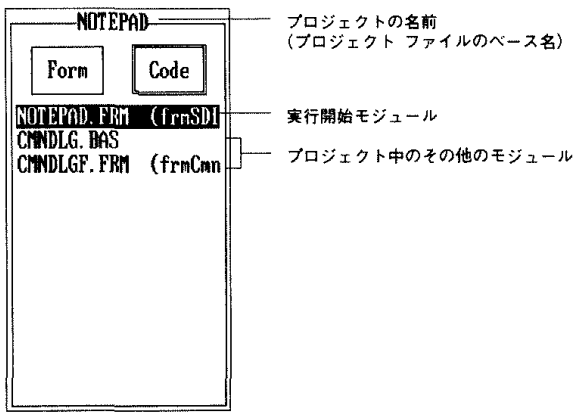
複数のフォームを使うアプリケーションでは、すべての **$FORM** メタコマンドを 1 つのインクルード ファイルにまとめておき、各モジュールのモジュール レベルに **$INCLUDE** メタコマンドを記述して、そのファイルをインクルードすると便利です。

```
'$INCLUDE: 'ALLFORMS.BI'
```

6 プロジェクトの管理

Visual Basic は、アプリケーションをプロジェクトとして扱います。プロジェクトは 1 つ以上のモジュール (ソース ファイル、つまりフォームまたはコード モジュール) で構成されます。特にアプリケーションが複数のモジュールで構成されているとき、プロジェクトとして扱うことによって、全体の管理が簡単になります。

アプリケーション、つまりプロジェクトが複数のモジュールで構成されているとき、Visual Basic は、プロジェクト ファイルと呼ばれる拡張子 .MAK のテキスト ファイルを作成します。このファイルには、そのプロジェクトに含まれるすべてのフォームおよびコード モジュールの名前が記録されます。これらのファイルの名前は、プログラミング環境のプロジェクト ウィンドウに表示されるものと同じです。

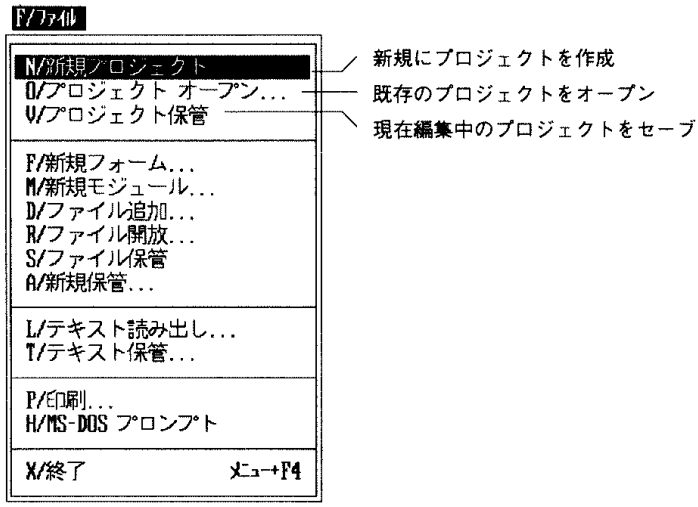


コマンドラインで特に指定せずに Visual Basic を起動すると、プロジェクト ウィンドウにはデフォルトのコード モジュールの名前である UNTITLED と表示されます。新規にフォームまたはコード モジュールを作成したりロードしたりすると、追加されたモジュールの名前はプロジェクト ウィンドウに表示されます。複数のモジュールで構成されるアプリケーションを最初にセーブしようとするとき、プロジェクト ファイル名の入力を求めるダイアログ ボックスが表示されます。以後、このアプリケーションはプロジェクトとして管理されます。Visual Basic を起動するときにコマンドライン引数としてプロジェクト ファイル名を指定すると、そのプロジェクトに含まれるすべてのモジュールがメモリにロードされます。また、プログラミング環境からプロジェクトをオープンするときにも同じです。さらに、実行可能 (.EXE) ファイルやライブラリを作成するときも、プロジェクトに含まれるすべてのフォームおよびコード モジュールから、実行可能 (.EXE) ファイルまたはライブラリが作成されます。プロジェクトとして管理することによって、別のディレクトリに記録されているモジュールをプロジェ

クト中に含めたり、あるモジュールを複数のプロジェクトで共有したりすることができます。

6.1　プロジェクトの作成，オープン，保管

プログラミング環境とフォーム デザイナでは、[ﾌｧｲﾙ] メニューの 3 つのコマンドを使って、プロジェクトを管理します。



▼ 新しいプロジェクトを作成するには

- [ﾌｧｲﾙ] メニューの [新規プロジェクト] コマンドを選択します。

  そのとき別のプロジェクトを編集中で、まだ保管していなければ、新しいプロジェクトを作成する前に、現在のプロジェクトを保管するかどうかをたずねるダイアログ ボックスが表示されます。新規にプロジェクトを作成すると、プログラミング環境から作成したときは UNTITLED.BAS という名前のコード モジュールが作成され、フォーム デザイナから作成したときは FORM1.FRM という名前のフォーム モジュールが作成されます。

▼ 既存のプロジェクトをオープンするには

1. [ﾌｧｲﾙ] メニューの [プロジェクト オープン] コマンドを選択します。

   [プロジェクト オープン] ダイアログ ボックスが現れ、カレント ディレクトリに含まれるすべてのプロジェクト (.MAK) ファイルの一覧が表示されます。

2. オープンする .MAK ファイルの名前を選択します。

同時に複数のプロジェクトをオープンすることはできませんが、あるプロジェクトから別のプロジェクトにフォームまたはコード モジュールをコピーすることはできます。

▼ Visual Basic 起動時に既存のプロジェクトをオープンするには

- コマンド ラインで **VBDOS** *projectname* と入力します。

  projectname は、オープンするプロジェクト ファイルのファイル名です。拡張子 .MAK は省略できます。Visual Basic が起動すると、プロジェクトに含まれるすべてのフォームおよびコード モジュールがメモリにロードされます。指定したプロジェクトが見つからないときは、同じベース名で拡張子 .BAS の新しいファイルが作成されます。

▼ プロジェクトを保管するには

- [ﾌｧｲﾙ] メニューの [プロジェクト保管] コマンドを選択します。

  プロジェクトを最初に保管するときは、プロジェクトの名前の入力を求めるダイアログ ボックスが表示されます。デフォルトでダイアログ ボックスに表示される名前は PROJECT.MAK です。そのプロジェクトの機能を表す、適切な名前を入力してください。既存のプロジェクトを保管するときは、ダイアログ ボックスは表示されません。

▼ 新規にフォームまたはコード モジュールを作成して、プロジェクトに追加するには

- フォーム モジュールを作成して追加する場合は [ﾌｧｲﾙ] メニューの [新規フォーム] コマンドを選択し、コード モジュールの場合は [新規モジュール] コマンドを選択します。

  フォーム デザイナからは、フォーム モジュールの追加だけが可能です。

  コマンドを選択したとき、それまで編集していたモジュールがまだ保管されていなければ、保管するかどうかをたずねるダイアログ ボックスが表示されます。また、新しく作成してプロジェクトに追加するフォームまたはコード モジュールの名前の入力を求めるダイアログ ボックスが表示されます。

重　要　最終的に実行可能 (.EXE) ファイルやライブラリを作成したいときは、コード モジュールとフォーム モジュールに、同じベース名を付けないように注意してください。例えば、コード モジュール NEW.BAS とフォーム モジュール NEW.FRM が同じプロジェクトに含まれるとき、この 2 つのファイルをコンパイルすると、両方とも NEW.OBJ という名前でオブジェクト ファイルを生成しようとするため、実行可能ファイルを作成できません。

▼ 既存のフォームまたはコード モジュールをプロジェクトに追加するには

- プログラミング環境では [ﾌｧｲﾙ] メニューの [ファイル追加] コマンドを選択し、フォーム デザイナでは [フォーム追加] コマンドを選択します。

  カレント ディレクトリに含まれるファイルの一覧がダイアログ ボックスに表示されるので、追加したいファイルの名前を選択してください。

  フォーム デザイナからは、フォーム モジュールの追加だけが可能です。

▼ **プロジェクトからフォームまたはコード モジュールを解放するには**

- プログラミング環境では [ﾌｧｲﾙ] メニューの [ファイル解放] コマンドを選択し、フォーム デザイナでは [フォーム解放] コマンドを選択します。

  プロジェクトに含まれているファイルの一覧がダイアログ ボックスに表示されるので、解放したいファイルの名前を選択してください。

  フォーム デザイナからは、フォーム モジュールの解放だけが可能です。

注　意　[ﾌｧｲﾙ] メニューの [ファイル解放] または [フォーム解放] コマンドは、指定したモジュールをプロジェクト ファイルに記録されているリストから削除するだけで、そのファイル自身をディスクから削除するわけではありません。

▼ **プロジェクトに含まれるフォームまたはコード モジュールを個々に保管するには**

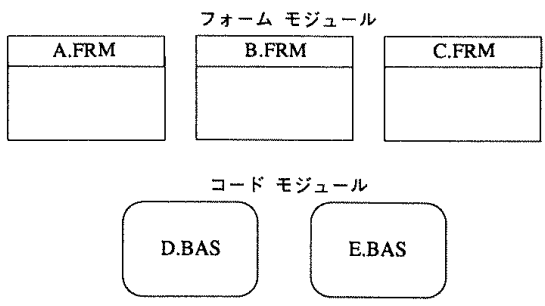
- プログラミング環境では [ﾌｧｲﾙ] メニューの [ファイル保管] または [新規保管] コマンドを選択し、フォーム デザイナでは [フォーム保管] または [新規保管] コマンドを選択します。

  これらのコマンドは、そのときアクティブになっている 1 つのモジュールを保管するだけです。プロジェクト全体を保管するには、[プロジェクト保管] コマンドを使います。また、そのモジュールを別の名前で保管したいときは、[新規保管] コマンドを選択します。プロジェクトでは、前の名前のファイルに代わって、新しく付けた名前のファイルが登録されます。

  [新規保管] コマンドでは、プロジェクトの名前の変更はできません。プロジェクトの名前を変更するには、いったん Visual Basic を終了して、MS-DOS の RENAME コマンドを使って .MAK ファイルのファイル名を変更してください。

実行可能ファイルの作成

コンパイルおよびリンクを行って、実行可能 (.EXE) ファイルを作成するときにも、プロジェクト単位で作業が行われます。複数のモジュールで構成されているアプリケーションの例を次に示します。



このプロジェクトの .MAK ファイルには、A.FRM, B.FRM, C.FRM, D.BAS, E.BAS の 5 つのファイル名が記録されています。

▼ プログラミング環境で実行可能ファイルを作成するには

1. [実行] メニューの [EXE ファイル作成] コマンドを選択します。

   [EXE ファイル作成] ダイアログ ボックスが表示されます。

2. 必要に応じてコンパイラ オプションを指定し、<作成> または <作成後終了> ボタンを選択します。

   コンパイラ (.BC.EXE) およびリンカ (LINK.EXE) が自動的に起動され、実行可能 (.EXE) ファイルが作成されます。出力画面には、コンパイラおよびリンカの実行状況が表示されます。

重 要 プロジェクトを構成する各モジュールには、別々のファイル名を指定する必要があります。特に、コード モジュールとフォーム モジュールに同じベース名を指定しないように注意してください。例えば、コード モジュール NEW.BAS とフォーム モジュール NEW.FRM が同じプロジェクトに含まれるとき、この 2 つのファイルをコンパイルすると、両方とも NEW.OBJ という名前でオブジェクト ファイルを生成しようとするため、実行可能ファイルを作成できません。

コマンド ラインから直接コンパイラやリンカを起動し、実行可能ファイルを作成することもできます。次のようなときに、この方法を使ってください。

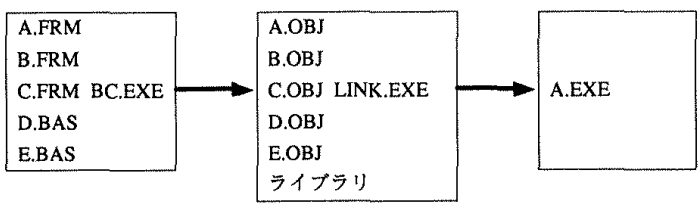
- 各モジュールに対して、別々のコンパイラ オプションを指定したいとき
- 他のプログラミング言語で記述したオブジェクト モジュールをリンクしたいとき
- プログラミング環境以外のテキスト エディタを使って、コードを書きたいとき
- NMAKE などのビルドを自動化するツールを使いたいとき

BC, LINK, LIB のコマンドライン オプションについては、オンライン ヘルプを参照してください。

▼ コマンド ラインからの実行可能ファイルを作成するには

1. コマンドラインで各モジュールに対して適切なコンパイル オプションを指定し、個別にコンパイルします。

2. 生成されるオブジェクト ファイルを Basic のランタイム ライブラリや他のライブラリとリンクして、実行可能ファイルを作成します。



プロジェクトのクローズ

Visual Basic を終了するときや、新しいプロジェクトの作成や既存のプロジェクトのオープンを行うとき、プロジェクトはクローズされます。このとき、編集中のプロジェクトがまだ保管されていなければ、保管するかどうかをたずねるダイアログ ボックスが表示されます。

6.2　Visual Basic のファイルとモジュール

Visual Basic のモジュールには、フォーム モジュール (.FRM ファイル) とコード モジュール (.BAS ファイル) の 2 種類があります。アプリケーションの作成には、これら 2 種類のモジュール以外にも、いくつかのファイルが使われます。通常、Visual Basic で作成するアプリケーションは、複数のファイルで構成されます。また、1 つのフォームまたはコード モジュールだけから構成される単純なアプリケーションを作成することも可能です。

Visual Basic のアプリケーションで使われるファイルには、次のようなものがあります。

表 6.1　Visual Basic のファイル/モジュールの種類

ファイルの種類	説明
コード モジュール (.BAS)	フォームに直接関連しない Basic コードの集まりです。フォームを使わないアプリケーションでは、コード モジュールだけでプロジェクトを構成します。フォームを使うアプリケーションでは、他のモジュールから利用するプロシージャを集めたり、実行開始モジュールとして初期化処理を行うために使います。
フォーム モジュール (.FRM)	フォームと関連する情報の集まりです。1 つのフォームに対して、1 つのフォーム モジュールが作成されます。フォームとフォーム上のコントロールに関する、Visual Basic が直接管理している情報が記録されています。また、そのフォームに関するイベント プロシージャが記録されます。必要に応じて、ジェネラル プロシージャを含めることもできます。

インクルード ファイル (.BI)	定数や変数の宣言，ユーザー定義型の定義，プロシージャの宣言などの集まりです。特に複数のモジュールで構成されるアプリケーションなどで、これらの宣言や定義をまとめて管理するために使います。インクルード ファイルの内容は、それぞれのモジュールのモジュール レベルで **$INCLUDE** メタコマンドが指定されている位置に展開されます。インクルード ファイルもアプリケーションを構成するファイルの 1 つですが、プロジェクトには含まれず、その名前はプロジェクト ファイルには記録されません。Visual Basic が提供しているインクルード ファイル CONSTANT.BI には、Visual Basic アプリケーションで一般的に使われる多くの定数の定義が含まれています。
プロジェクト ファイル (.MAK)	プロジェクトを構成するすべてのモジュールのファイル名が記録されています。
オブジェクト モジュール ライブラリ ファイル (.LIB)	実行可能ファイルを作成するときに、フォームおよびコード モジュールをコンパイルして作成されたオブジェクト ファイル (.OBJ) とリンクされます。
Quick ライブラリ ファイル (.QLB)	プログラミング環境にロードして使うことができるライブラリです。通常、 Quick ライブラリに含まれるプロシージャの宣言をまとめたインクルード ファイルと組み合わせて使います。

コード モジュール

コード モジュールは、フォームに直接関連しない Basic コードのみを含むファイルです。コード モジュールの内部は、次の 2 つの部分に分けることができます。

- モジュール レベル コード。プロシージャに含まれない部分をモジュール レベルと呼び、そこに記述されたコードをモジュール レベル コードと呼びます。コード モジュールのモジュール レベルには、宣言や定義のほかに，実行可能ステートメントやエラー処理ルーチンを記述できます。コード モジュールがアプリケーションの実行開始モジュールに設定されているときは、モジュール レベル コードが最初に実行されます。
- プロシージャ レベル コード。プロシージャ中に記述されたコードが、プロシージャ レベル コードです。プロシージャには、**SUB** プロシージャと **FUNCTION** プロシージャの 2 種類があります。1 つのモジュールに、複数のプロシージャを記述することができます。

フォーム モジュール

フォーム モジュールは、フォームに関連する情報を集めたファイルです。プロパティの設定など、フォームやコントロール自身を表示，管理するために、Visual Basic が直接管理している情報も含まれています。

プログラマから見ると、フォーム モジュールの内容は、コード モジュールと同様に 2 つの部分に分けることができます。ただし、プロシージャ レベル コードは、さらにイベント プロシージャとジェネラル プロシージャの 2 つに分けられます。

- モジュール レベル コード。コード モジュールとは異なり、フォーム モジュールのモジュール レベルには宣言，定義，メタコマンドのみを記述します。実行可能ステートメントを記述することはできません。また、フォーム モジュールではモジュール レベルのエラー処理ルーチンは設定できません。
- イベント プロシージャ。イベントが発生したときに自動的に呼び出されるプロシージャです。イベント プロシージャも、プロシージャ レベル コードの一部です。また、イベント プロシージャは **SUB** プロシージャの一種です。
- ジェネラル プロシージャ。プロシージャ レベル コードのうち、イベント プロシージャではないものです。**SUB** プロシージャと **FUNCTION** プロシージャの 2 種類にさらに分類できます。

注　意　フォーム モジュールのプロシージャは、そのモジュール内でのみ有効です。他のモジュールから呼び出すことはできません。

バイナリ形式とテキスト形式

フォームおよびコード モジュールを保管する形式には、バイナリ形式とテキスト形式があります。

バイナリ形式は中間コードと呼ばれる形態でファイルに記録するもので、Visual Basic だけが認識可能であり、市販のテキスト エディタなどで編集することはできません。バイナリ形式は、ロードやセーブの時間が速く、特に大規模なモジュールではファイルサイズも小さくできます。フォーム モジュールは、デフォルトではバイナリ形式で保管されます。

一方テキスト形式は、通常のテキスト ファイルと同じように文字コードだけで内容を表現したファイルです。テキスト形式で保管すると、市販のテキスト エディタで編集したり、テキスト ファイルが扱えるツールを使って処理を行うことなどができます。コード モジュールは、デフォルトではテキスト形式で保管されます。

フォーム モジュールもコード モジュールも、保管時にダイアログ ボックスでファイルの形式を指定することができます。どちらの形式で保存しても、拡張子などは同じです。またロード時には、Visual Basic がどちらの形式であるかを自動的に判断して読み込むため、特に指定する必要はありません。

あるモジュールがどちらの形式で保管されるかは、最初に保管するときの指定によって決まります。デフォルト以外の形式を指定した場合、特に指定しなければ、その後の保管時にも自動的にその形式で保管されます。

▼ **保管するファイルの形式を指定するには**

- 保管を行うときに表示されるダイアログ ボックスで、ファイル形式の欄からバイナリ形式またはテキスト形式のオプション ボタンを選択します。

▼ **すでにバイナリ形式で保管しているモジュールを、テキスト形式で別ファイルに保管するには**

- [ﾌｧｲﾙ] メニューの [新規保管] コマンドを選択し、テキスト形式のオプション ボタンを選択します。この方法でフォーム モジュールを保管すると、フォームの外観やフォーム上に配置されたコントロールの情報もテキスト形式に変換されて保管されます。詳しくは、付録 A 「ASCII フォームのフォーマット」を参照してください。

注 意 QuickBasic や Microsoft Basic で作成したファイル (.BAS ファイル) を Visual Basic で読み込むには、これらのファイルはテキスト形式で保管されている必要があります。詳しくは『機種依存情報』を参照してください。

インクルード ファイル

インクルード ファイルは、変数や定数, フォームなどの宣言や、ユーザー定義型の定義などを集めた特別なテキスト ファイルです。複数のモジュールで同じ宣言や定数を使うときに役立ちます。インクルード ファイルの拡張子は .BI です。

注 意 インクルード ファイルに、実行可能ステートメントを記述することは避けてください。

Visual Basic では、次の 2 つの汎用インクルード ファイルが提供されています。

- CONSTANT.BI。色や ControlPanel プロパティの要素を示す定数など、Visual Basic で作成するアプリケーションで共通して使われる宣言を含みます。
- VBDOS.BI。**Absolute, Interrupt, InterruptX** ルーチンなどを含む Quick ライブラリ VBDOS.QLB を利用するときに使うインクルード ファイルです。

インクルード ファイルをインクルードするには、**$INCLUDE** メタコマンドを使います。

'$INCLUDE: '*include_file*'

メタコマンドの前にコメントを表すシングル クォーテーション (') が付いていることに注意してください。**$INCLUDE** などのメタコマンドは、必ずモジュール レベルに置かなければなりません。また、すべての実行可能ステートメントの前に置いてください。

例えば、CONSTANT.BI をインクルードするときは、モジュール レベルに次のメタコマンドを挿入します。

```
'$INCLUDE: 'CONSTANT.BI'
```

ライブラリ ファイル

ライブラリはオブジェクト コードを集めたもので、アプリケーションとリンクできる状態にあるコンパイル済みのルーチンが含まれます。

Visual Basic は次の 2 種類のライブラリをサポートします。

ファイルの種類	説明
.LIB	オブジェクトモジュール ライブラリ。コンパイルして生成されたオブジェクト ファイルとリンクして、実行可能ファイルまたは Quick ライブラリを作成します。
.QLB	Quick ライブラリ。特殊なライブラリで、プログラミング環境にロードして使います。

実行可能 (.EXE) ファイルの実行形態

Visual Basic で作成する実行可能 (.EXE) ファイルには、次の2 種類があります。どちらが生成されるかは、リンクされるライブラリによって決まります。

- ランタイム分離型実行可能ファイル。この形式の実行可能ファイル (アプリケーション) を実行するときは、ランタイム モジュールと呼ばれる特別なファイル (VBDRT10.EXE) が必要です。Visual Basic が最初から用意しているステートメントや関数の実行ルーチン (ランタイム ルーチンと呼びます) は、実行可能ファイルにではなく、このランタイム モジュールに含まれています。このタイプの実行可能ファイルを作成するには、[EXE ファイル作成] ダイアログ ボックスで [ランタイム分離型] オプションを指定します。コマンド ラインからコンパイルするときは、/O コンパイラ オプションを指定せずにコンパイルし、ランタイム分離型のライブラリである VBDRT10.LIB とリンクします。
- 独立型実行可能ファイル。この形式の実行可能ファイルは、ランタイム モジュールが必要なく、単独で実行できます。ランタイム ルーチンも、すべて実行可能 (.EXE) ファイル中に含まれます。独立型実行可能ファイルを作成するには、[EXE ファイル作成] ダイアログ ボックスで [独立型] オプションを指定します。コマンド ラインからコンパイルするときは /O コンパイラ オプションを指定してコンパイルし、独立型のライブラリである VBDCL10.LIB とリンクします。

独立型実行可能ファイルには、ライブラリ VBDRT10.EXE に含まれているものと同等のランタイム ルーチンがリンクされるため、ランタイム 分離型実行可能ファイルよりもサイズが大きくなります。実行可能ファイルの数が多いときは、VBDRT10.EXE を使ってコードを共有すると、ファイルのサイズを小さくできます。

Quick ライブラリ

Quick ライブラリは、プログラミング環境でインタープリタを使って実行するときに使う特殊なライブラリです。Visual Basic の起動時に、/L オプションで指定してプログラミング環境にロードします。汎用的なプロシージャをモジュールに集めて Quick ライブラリを作成しておけば、それらのプロシージャを、Visual Basic が最初から用意しているステートメントや関数と同じように扱うことができます。

実行可能ファイルを作成するためには、同じ内容を持つオブジェクト モジュール ライブラリ (.LIB) が必要です。通常、同じ内容の Quick ライブラリとオブジェクト モジュール ライブラリを組にして作成，管理します。

例えば、Visual Basic が提供している、アセンブリ言語ルーチンなどを含む Quick ライブラリ VBDOS.QLB には、対応するオブジェクト モジュール ライブラリとして VBDOS.LIB が用意されています。

Quick ライブラリについて詳しくは、第 18 章「Quick ライブラリの作成」を参照してください。

テキスト ファイル

Visual Basic では、Basic コード以外のデータを含む、通常のテキスト ファイルも扱うことができます。テキスト ファイルのデフォルトの拡張子は .TXT です。この機能は、プログラミング環境で、README のようなドキュメント ファイルを作成するときに便利です。テキスト ファイルは、プロジェクト (.MAK) ファイルには記録されません。

▼ **新しいテキスト ファイルを作成するには**

1. [ﾌｧｲﾙ] メニューの [新規モジュール] コマンドを選択します。

   [新規モジュール] ダイアログ ボックスが表示されます。

2. ファイル形式の欄から、テキストを示すオプション ボタンを選択します。

▼ **テキスト ファイルをロードするには**

1. [ﾌｧｲﾙ] メニューの [ファイル追加] コマンドを選択します。

   [ファイル追加] ダイアログ ボックスが表示されます。

2. ファイル形式の欄から、テキストを示すオプション ボタンを選択します。

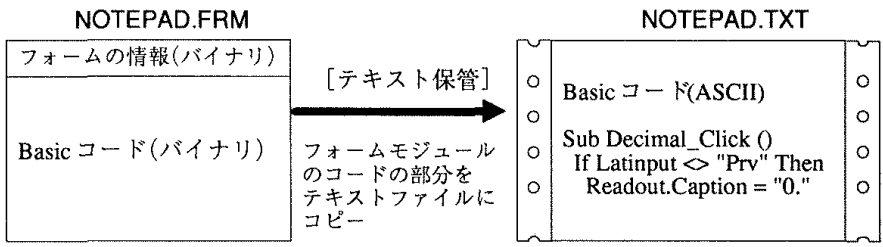
README ファイルのように、Basic コード以外のテキストを含むファイルをロードするときは、テキスト ファイルであることをオプションで指定してプログラミング環境にロードする必要があります。このようにすると、構文チェックが行われず、プロジェクト ファイルにも追加されません。テキスト ファイルとしてロードすると、プログラミング環境をテキスト エディタとして使ってファイルを編集できます。

▼ **フォームまたはコード モジュールの内容をテキスト ファイルとして保管するには**

1. プログラミング環境で、目的のモジュールのコードが表示されているコード ウィンドウを選択します。
2. [ﾌｧｲﾙ] メニューの [テキスト保管] コマンドを選択します。

   [テキスト保管] ダイアログ ボックスが表示されます。
3. 作成するテキスト ファイルのファイル名を入力します。

フォーム モジュールの内容をテキスト ファイルとして保管するときは、フォーム上のコントロールなどに関するデータは無視され、Basic コードのみが書き込まれます。



▼ **Basic コードを含むテキスト ファイルの内容を、現在編集中のモジュールに追加するには**

1. プログラミング環境でコード ウィンドウを選択し、テキスト ファイルの内容を挿入する位置にカーソルを移動します。
2. [ﾌｧｲﾙ] メニューの [テキスト読み出し] コマンドを選択します。
3. 読み込むテキスト ファイルの名前を入力します。
4. <追加> ボタンをクリックします。

   テキスト ファイルの内容が、Basic コードとしてカーソル位置に挿入されます。

このようにすると、テキスト ファイルの内容は Basic コードとして扱われ、読み出しと同時に構文チェックが行われます。

▼ **Basic コードを含むテキスト ファイルの内容読み出し、現在編集中のモジュールと置き換えるには**

1. プログラミング環境で、更新するモジュールが表示されているコード ウィンドウを選択します。
2. [ﾌｧｲﾙ] メニューの [テキスト読み出し] コマンドを選択します。
3. 読み込むテキスト ファイルの名前を入力します。
4. <置換> ボタンをクリックします。

   テキスト ファイルの内容が、Basic コードとして解釈され、編集していたモジュールの内容と置き換えられます。

6.3 実行開始ファイルの設定

複数のモジュールで構成されるプロジェクトでは、モジュールのうちの1つを“実行開始モジュール”として設定する必要があります。アプリケーションの実行は、この実行開始モジュールから始まります。実行開始ファイルには、次のような特徴があります。

- デフォルトでは、そのプロジェクトで最初に作成したモジュールが、そのプロジェクトの実行開始モジュールになります。
- 実行開始モジュールとして、フォーム モジュールとコード モジュールのどちらを指定してもかまいません。
- 実行開始モジュールがコード モジュールのときは、そのモジュールのモジュール レベル コードの最初の実行可能ステートメントから実行が開始されます。
- 実行開始モジュールがフォーム モジュールのときは、フォームをロードした後、イベントが発生するのを待ちます。

注 意 コマンドラインからリンカ (LINK.EXE) を直接呼び出してリンクするときは、実行開始モジュールを LINK コマンド ラインのオブジェクト フィールドの最初に指定する必要があります。

▼ **実行開始モジュールを変更するには**

1. [実行] メニューの [実行開始ファイル設定] コマンドを選択します。

   現在ロードされているモジュールのリストが表示されます。

2. リストから新しい実行開始モジュールを選択します。

6.4 フォームとコードの印刷

Visual Basic には、プロジェクトに含まれるソース コードと、フォームおよびコントロールの情報を出力する機能があります。

▼ **コードの印刷をするには**

1. プログラミング環境で、[ﾌｧｲﾙ] メニューの [印刷] コマンドを選択します。

   [印刷] ダイアログ ボックスが表示されます。

2. 印刷範囲や出力先などのオプションを選択して、<了解> ボタンをクリックします。

▼ **フォームの表示イメージと、フォームとコントロールの各プロパティの設定の一覧を出力するには**

1. フォーム デザイナで、[ﾌｧｲﾙ] メニューの [印刷] コマンドを選択します。

   [印刷] ダイアログ ボックスが表示されます。

2. 印刷範囲や出力先などのオプションを選択して、<了解> ボタンをクリックします。

[印刷] ダイアログ ボックスのオプションについて詳しくは、オンライン ヘルプを参照してください。

7 テキストの処理

Visual Basic は、アプリケーションでテキストを扱うために、次のようなオブジェクトを用意しています。

- テキスト ボックス。テキスト ボックスに表示されているテキストは、アプリケーション実行中にユーザーが編集できます。
- フォームとピクチャー ボックス。フォームやピクチャー ボックスに表示されるテキストはアプリケーション実行中にユーザーが変更することはできません。
- **CLIPBOARD** オブジェクト。テキストのカット，コピー，ペーストに使います。
- **PRINTER** オブジェクト。テキストをプリンタやファイルに出力するために使います。

この他にも、Caption プロパティを使うと、ラベルやコマンド ボタン上にもテキストを表示できます。

7.1 テキスト ボックスの使い方

テキスト ボックスは、次のように多くの機能を持ったコントロールです。

- 複数行が表示可能なテキスト ボックスとワードラップ
- テキスト ボックス中のデータが有効かどうかの確認
- キーストロークの変更と制限
- 小数点の変更

複数行のテキスト ボックスとワードラップ

テキスト ボックスの外観と動作は、主に MultiLine と ScrollBars の 2 つのプロパティで決まります。

プロパティ	機能	設定
MultiLine	テキスト ボックスに複数行のテキストを入力または表示できるかどうかを決めます。実行時は読み込み専用です。	True (-1) False (0)

ScrollBars	テキスト ボックスに水平および垂直スクロール バーを付けるかどうかを決めます。実行時は読み込み専用です。	0 - なし (デフォルト) 1 - 水平 2 - 垂直 3 - 両方

注　意　ScrollBars プロパティは、垂直および水平スクロール バー コントロールとは異なります。スクロール バー コントロールは独立したコントロールであり、スクロール バー コントロールを使って、テキスト ボックスにスクロール バーを付けることはできません。

複数行のテキストの表示

通常、テキスト ボックスに表示できるテキストは 1 行だけです。ただし、MultiLine プロパティを True (-1) に設定しておくと、実行時にテキスト ボックスに対して複数行のテキストを入力したり表示したりできるようになります。MultiLine プロパティはデザイン時にのみ設定可能です。

テキスト ボックスでのワード ラップ制御

複数行のテキスト ボックスでは、水平スクロール バーが付けられていないと、自動的にワード ラップが行われます。ユーザーは、テキスト ボックスに入力するとき、改行のために [ENTER] キーを押す必要はありません。テキストが長くて 1 行で表示できないときは、次の行に自動的にラップされます。

水平スクロール バーが付けられている複数行のテキスト ボックスでは、ワードラップは行われません。改行するには [ENTER] キーを押します。1 行には、最大 32K のテキストを入力できます。

複数行のテキスト ボックスでの改行

MultiLine プロパティが True (-1) に設定されているときでも、デザイン時には Text プロパティに複数行のテキストを設定することはできません。コード中からテキスト ボックスに文字を挿入するときには、改行はキャリッジ リターンとラインフィード (ASCII 文字 13 と 10) によって示します。例えば次のイベント プロシージャは、フォームをロードするときに、複数行のテキスト ボックス (Text1) に 2 行のテキストを挿入しています。

```
SUB Form_Load
    NL$ = CHR$(13) + CHR$(10)          '改行を示す変数を定義します.
    Text1.Text = "このテキスト ボックスには" + NL$ + "2 行あります"
END SUB
```

実行時にユーザーがテキスト ボックスに対して改行を挿入するとき、デフォルトのコマンド ボタンがあるかどうかによって操作が異なります。デフォルトの (Default プロパティが True (-1) に設定されている) コマンド ボタンを含むフォームでは、ユーザーが [ENTER] キーを押すと、そのコマンド ボタンに対する Click イベントが発生します。この場合、改行したいときは [CTRL] + [ENTER] キーを押します。デフォルトのコマンド ボタンがなければ、[ENTER] キーはテキスト ボックスの改行と解釈されます。

テキスト ボックスに入力されたデータの処理

KeyPress イベントに対するイベント プロシージャを用意すると、入力できる文字を制限したり、入力された文字を変換することができます。このような処理を行うときは、KeyPress イベントを使う方法がもっとも簡単です。KeyDown および KeyUp イベントとは異なり、KeyPress イベントは文字コード (ASCII および漢字連続コード) で定義されている文字が入力されたときのみ発生し、ファンクション キーや方向キーなどの特殊キーでは発生しません。文字コードに定義されている文字とは、2 バイト文字を含むすべての印字可能文字と，[CTRL] キーと A ～ Z の文字の組み合わせ，[ENTER] や [BS] キーなどのいくつかの標準文字が含まれます。

KeyPress イベントに対するイベント プロシージャは、引数 *KeyAscii* をとります。ユーザーがキーを押してイベント プロシージャが自動的に呼び出されるとき、ユーザーが押したキー、または日本語入力システムを通して入力した文字に対応する文字コードが、パラメータとして指定した変数に渡されます。イベント プロシージャ内では、この変数の値をチェックして、ユーザーがどの文字を入力したのかを知ることができます。この引数は参照で渡されます。つまり、プロシージャ中でこの変数の値を変更すると、テキスト ボックスには、その値に対応する文字が表示されます。

なお、文字そのものではなく、対応する文字コードが数値として渡されることに注意してください。漢字連続コードは、JIS 漢字またはシフト JIS コードの、コードが不連続になっているところを詰め、2 バイト文字の最初の文字である 2 バイトのスペースを 256 として、連続して並べたコードです。コードの大小関係は、JIS 漢字またはシフト JIS コードと同じです。

KeyPress イベントに対する典型的なイベント プロシージャを次に示します。

1. イベント プロシージャが呼び出されたら、KeyAscii パラメータとして指定した変数の値を、**CHR$** 関数を使って 1 文字の文字列に変換します。
2. この文字列に対して操作やテストを行います。関係演算子 (=, >, <, <=, >=) を使うと、文字コードに従って大小関係を判断できます。
3. アルファベットの大文字小文字を変換するなど、テキスト ボックスに表示する文字を変えたいときは、その文字を **ASC** 関数で文字コードに変換して、パラメータとして指定した変数に代入します。また、ユーザーのキー入力を無効にしたいときは、変数に 0 を代入します。

多くのオブジェクトは KeyPress イベントを認識しますが、入力された文字を表示できるのはテキスト ボックスとコンボ ボックスだけです。したがって、ステップ 3 はテキスト ボックスとコンボ ボックスに対してのみ意味を持ちます。

参　考　Visual Basic の文字列関数について詳しくは、オンライン ヘルプまたは『リファレンス』を参照してください。また、漢字連続コードについては、オンライン ヘルプを参照してください。

大文字への変換

次の KeyPress イベント プロシージャは、ユーザーがテキスト ボックス Text1 に対して入力したアルファベットの小文字を大文字に変換します。

```
SUB Text1_KeyPress (KeyAscii AS INTEGER)
    C$ = CHR$(KeyAscii)                 '文字コードを文字に変換します.
    KeyAscii = ASC(UCASE$(C$))          '文字を大文字に変換します.
END SUB
```

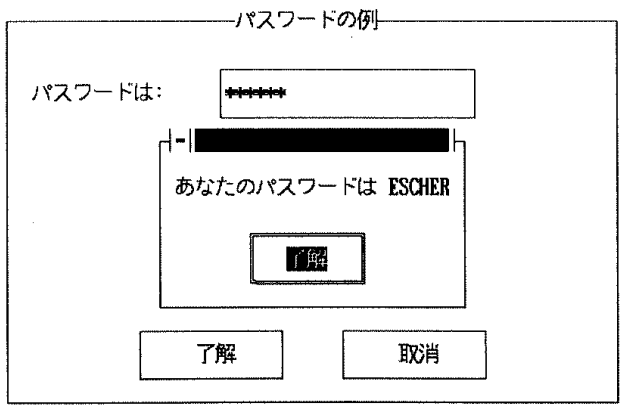
まず、渡された文字コードを **CHR$** 関数で実際の文字に変換してから、**UCASE$** 関数で大文字に変換し、**ASC** 関数で再び文字コードに戻し、KeyAscii 変数に代入しています。**UCASE$** 関数は、2 バイト文字の小文字も大文字に変換しますが、小文字以外の文字に対しては何も行いません。

特定の文字の無視

次の例は、ユーザーが入力した文字が特定の文字であるとき、その入力を無効にし、テキスト ボックスに表示されないようにします。

ユーザーは、1 バイトのアルファベットまたは数字だけで構成された 8 文字以下のパスワードをテキスト ボックスに入力し、<了解> をクリックするか、[ENTER] キーを押します。間違った文字を入力したり、9 文字以上入力すると、アプリケーションはビープ音を鳴らして文字を受け付けません。有効な文字の ASCII コードは次のようになります。

- 48 ～ 57 (1 ～ 9)
- 65 ～ 90 (A ～ Z)
- 97 ～ 122 (a ～ z)



正しいパスワードを入力すると、パスワードを含むダイアログ ボックスが表示されます。一般にパスワードは、セキュリティの面から画面には表示されません。このプログラムは、あくまで文字を変換するサンプルと考えてください。

▼ アプリケーションを作成するには

1. フォーム デザイナで、上の図に示すようなラベル，テキスト ボックス，2 つのコマンド ボタンをフォームに配置します。

2. 次の表に従って、オブジェクトのプロパティを設定します。

オブジェクト	プロパティの設定	使用するイベント
フォーム	Caption = パスワードの例	
ラベル	CtlName = lblPassWord Caption = パスワードは:	
テキスト ボックス	CtlName = txtPassWord Text = (なし)	KeyPress
上の コマンド ボタン	CtlName = cmdOK Default = True Caption = 了解	Click
下のコマンド ボタン	CtlName = cmdCancel Caption = 取消	Click

3. txtPassWord_KeyPress イベント プロシージャに次のコードを追加します。

```
SUB txtPassWord_KeyPress (KeyAscii AS INTEGER)
STATIC Temp AS STRING
DIM CharIn AS STRING                '入力された文字を保存する変数
    IF LEN(Temp) > 7 THEN           '9 文字以上入力すると，
        KeyAscii = 0                '入力された文字を無視します．
        BEEP
    '有効な文字か数字であることを確認します．
    ELSEIF (KeyAscii > 47 AND KeyAscii < 58) OR (KeyAscii >
64 AND KeyAscii < 91) OR (KeyAscii > 96 AND KeyAscii < 123)
THEN
        'KeyAscii を大文字に変換します．
        CharIn = UCASE$(CHR$(KeyAscii))
        'テキスト ボックスにアスタリスクを表示します．
        Temp = Temp + "*"
        txtPassWord.Text = Temp
        KeyAscii = 0
        '入力された文字をフォームの Tag プロパティに保持します．
        Tag = Tag + CharIn
    '文字または数字が入力されないとき
    ELSE
    KeyAscii = 0
    BEEP
    END IF
END SUB
```

4. cmdOK_Click イベント プロシージャに次のコードを追加します。

```
SUB cmdOK_Click ()
    msg$ = "あなたのパスワードは " + Tag
    MSGBOX msg$        '入力されたパスワードを表示します．
    SCREEN.HIDE
    END
END SUB
```

5. cmdCancel イベント プロシージャに次のコードを追加します。

```
SUB cmdCancel_Click ()
    SCREEN.HIDE
    END
END SUB
```

日本語の扱い

ASCII コードで定義されている文字はすべて 1 バイトで表すことができますが、漢字など JIS 漢字コード (またはシフト JIS コード) で定義されている文字 (いわゆる全角文字) は 2 バイトで表されています。Visual Basic では、2 バイト文字を扱うためのいくつかの関数が用意されています。

例えば **LEN** 関数は、引数に指定した文字列のバイト数を返します。指定した文字列がすべて ASCII 文字で構成されている場合は、**LEN** 関数が返す値は同時に文字数も表しますが、文字列に 2 バイト文字が含まれている場合は、文字数とバイト数は一致しません。ある文字列の文字数を調べるには、**KLEN** 関数を使います。

この他にも、2 バイト文字を扱うための関数およびステートメントには、次のようなものがあります。

関数およびステートメント	機能
CDBL$** 関数，**CSNG$ 関数	1 バイト系と 2 バイト系の文字を変換します。
KENT$ 関数	指定した文字列の中から 1 バイトまたは 2 バイトの文字だけを抜き出して返します。
KLEN 関数	指定した文字列の文字数を返します。
KPOS 関数	文字列中の指定した文字位置までのバイト数を返します。
KMID$ 関数	文字列の中から、指定した文字数分の文字列を返します。
KMID$ ステートメント	文字列変数の一部を別の文字列に置き換えます。
KINSTR 関数	ある文字列の中から別の文字列を検索し、最初に見つかった位置を返します。
JIS$** 関数，**KTN$ 関数	指定した文字列の最初の文字の JIS 漢字コードまたは区点コードを返します。
INPUT¥ 関数	指定したファイルから指定した文字数分の文字列を読み込んで返します。

日本語入力システムの制御

フォームが表示されている状態では、フォーカスがテキスト ボックスまたはコンボボックスにあるときだけ、日本語入力システムをオンにすることができます。また、**INPUTBOX$** 関数でダイアログ ボックスを表示しているときも、日本語入力システムをオンにできます。

ユーザーに入力を求めるとき、必ず 2 バイト文字が入力される場合は、あらかじめ日本語入力システムをオンにしておくと便利です。SetFEPMode および GetFEPMode%

ルーチンを使うと、コード中から日本語入力システムのオン/オフを制御したり、そのときの状態を取得したりできます。

これらはルーチンとして用意されているため、利用するときは必ず **DECLARE** ステートメントであらかじめ宣言しておかなければなりません。また、プログラミング環境上で利用するには、VBDOS 起動時に /L オプションを指定し、何らかの Quick ライブラリがロードされている必要があります。使い方などについて詳しくは、『リファレンス』またはオンライン ヘルプを参照してください。

文字列に含まれるダブル クォーテーションの扱い

文字列の中に 1 バイトのダブル クォーテーション (" ") が含まれることもあります。例えば次の文章を表示したいとします。

```
Visual Basic は略して "VB" と呼ばれることがあります。
```

文字列定数の中にダブル クォーテーションを含めるためには、次のようにダブル クォーテーションを 2 つ続けます。

```
msg$ = "Visual Basic は略して ""VB"" と呼ばれることがあります。"
```

CHR$ 関数で、ダブル クォーテーションを示す ASCII コード (34) を指定しても同じ意味になります。

```
msg$ = "Visual Basic は略して " + CHR$(34) + "VB" + CHR$(34)
  ⇒ + " と呼ばれることがあります。
```

7.2 CLIPBOARD オブジェクトの使い方

CLIPBOARD オブジェクトは、Visual Basic であらかじめ定義されている 3 つの特殊オブジェクトのうちの 1 つで、テキストのカット & ペーストに使うクリップボードを示しています。テキスト ボックスやコンボ ボックスのテキスト ボックス部分には、最初からカット & ペーストの機能が用意されています。切り抜きは [CTRL] + [X] および [SHIFT] + [DEL]、複写は [CTRL] + [C] および [CTRL] + [INS]、貼り付けは [CTRL] + [V] および [SHIFT] + [INS] に割り当てられています。単にテキストのカット & ペーストを行うだけであれば、特にコードを記述する必要はありません。

切り抜きまたは複写されたテキストは、**CLIPBOARD** オブジェクトに格納されます。また、貼り付けを行うときは、**CLIPBOARD** オブジェクトの内容が貼り付けられます。**CLIPBOARD** オブジェクトに保持できるテキストは 1 度に 1 つだけです。別のテキストを切り抜きまたは複写すると、元の内容は失われます。

CLIPBOARD オブジェクトにはプロパティや対応するイベントはありませんが、文字列変数やプロパティとの間でテキスト データをやり取りするためのメソッドが用意されています。これらのメソッドを利用すると、コード中からクリップボードの内容を操作できます。

次の図に示すように、**SETTEXT** メソッドと **GETTEXT** メソッドを使うと、クリップボードとの間で文字列データをやり取りします。

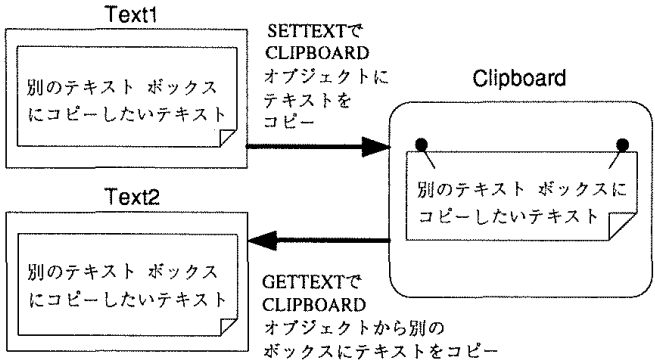


図 7.1　SETTEXT および GETTEXT メソッドによるクリップボードの操作

テキストの複写と貼り付け

クリップボードとの間でテキストをやり取りするには **SETTEXT**、**GETTEXT** メソッドを使います。**SETTEXT** メソッドは、テキストを **CLIPBOARD** オブジェクトに格納します。**GETTEXT** メソッドは、クリップボードに格納されているテキストを取り出します。

SETTEXT メソッドの構文を次に示します。

CLIPBOARD.SETTEXT *data$*

引数 *data$* はクリップボードに格納する文字列データです。

GETTEXT メソッドの構文を次に示します。

destination$ = **CLIPBOARD.GETTEXT** ( )

変数 *destination$* は取り出したテキストを格納する文字列変数またはプロパティです。**GETTEXT** メソッドは、関数のように式の中で使うことに注意してください。

テキストの選択

テキスト ボックスとコンボ ボックスは、**CLIPBOARD** オブジェクトを活用するときに便利な 3 つのプロパティを持っています。これらのプロパティは、コード中からテキストの選択 (反転表示) 状態を操作するために使います。実行時にユーザーがマウスやキーボードでテキストを操作すると、これらのプロパティの値も変化します。

プロパティ	説明
SelStart	反転表示されているテキストの先頭の位置を整数で示します。また、反転表示されているテキストがないときは 、次に入力される文字が挿入される位置を示します。 0 が先頭を示し、正の整数が先頭からの文字数を示します。表示されているテキストの文字数と同じ値が、コントロールの最後の文字の直後を表します。
SelLength	反転表示されているテキストの文字数を整数で示します 。

SelText	反転表示されているテキストを示します。反転表示されているテキストがないときは、このプロパティの値は null 文字列 ("") です。

コード中から SelStart および SelLength プロパティを設定すると、テキストを反転表示させることができます。例えば次のステートメントは、テキスト ボックスのすべてのテキストを反転表示させます。

```
Text1.SelStart = 0                    '最初の文字の直前から選択を開始します.
Text1.SelLength = KLEN (Text1.Text) 'テキストの最後まで選択します.
```

この例で、**KLEN** 関数はテキスト ボックス (Text1) に表示されているテキストの文字数を返します。この値を SelLength プロパティに代入すると、テキスト ボックスのすべての文字が反転表示されます。反転表示されたテキストは、自動的に SelText プロパティに代入されます。

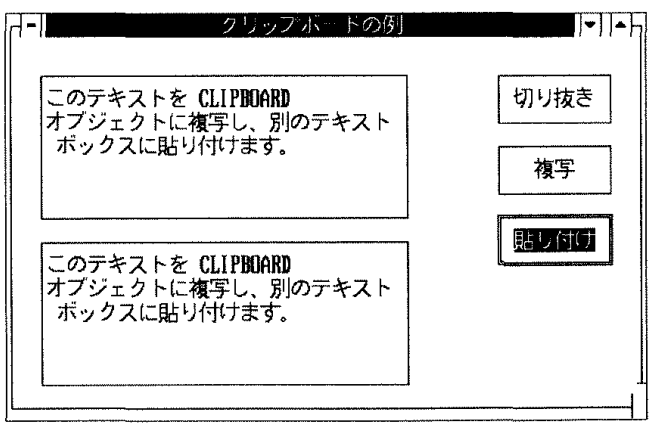
新しい文字列を SelText に代入すると、それまで反転表示されていたテキストが置き換えられ、元のテキストは失われます。また、反転表示は解除され、新しく挿入されたテキストの直後に挿入位置が移動します。例えば次のステートメントを実行すると、反転表示されていたテキストが I've just been inserted! に置き換えられます。

```
Text1.SelText = "I've just been inserted!"
```

反転表示されたテキストがないときは、指定した文字列が挿入位置に挿入されます。

テキスト ボックス用のカット & ペースト機能の例

ここでは、**SETTEXT** メソッドと **GETTEXT** メソッドを SelStart, SelLength, SelText プロパティと組み合わせて、コマンド ボタンを使ったテキスト ボックス用のコピー & ペースト機能を考えてみます。なお、テキスト ボックスにはキーボードを使ったカット & ペーストの機能はあらかじめ用意されています。



▼ **上のテキスト ボックスから下のテキスト ボックスに対してカット & ペーストを行うアプリケーションを作成するには**

1. フォーム デザイナで、上の図に示すように、フォームに 2 つのテキスト ボックスと 3 つのコマンド ボタンを配置します。
2. 次の表に従って、オブジェクトのプロパティを設定します。

オブジェクト	プロパティの設定	使用するイベント
フォーム	Caption = クリップボードの例	
上のテキスト ボックス	CtlName = Text1 Multiline = True (-1)	
下のテキスト ボックス	CtlName = Text2 Multiline = True (-1)	
上のコマンド ボタン	CtlName = cmdCut Caption = 切り抜き	Click
中央のコマンド ボタン	CtlName = cmdCopy Caption = 複写	Click
下のコマンド ボタン	CtlName = cmdPaste Caption = 貼り付け	Click

3. 3 つのコマンド ボタンの Click イベント プロシージャに次のコードを追加します。

```
SUB cmdCut_Click ()          '<切り抜き> ボタン
    CLIPBOARD.SETTEXT Text1.SelText
        '選択したテキストを CLIPBOARD にコピーします.
    Text1.SelText = ""
        'テキスト ボックスのテキストをクリアします.
END SUB

SUB cmdCopy_Click ()         '<複写> ボタン
    CLIPBOARD.SETTEXT Text1.SelText
        '選択したテキストを CLIPBOARD にコピーします.
END SUB

SUB cmdPaste_Click ()        '<貼り付け> ボタン
    Text2.SelText = CLIPBOARD.GETTEXT()
        'CLIPBOARD のテキストを Text2 にペーストします.
END SUB
```

参　考　テキストのカット，コピー，ペーストは、コマンド ボタンではなくメニューを使って行うこともできます。詳しくは、第 8 章「メニューの作成」を参照してください。

両方向への改造

ここまで行った作業で利用できるカット & ペーストは上のテキスト ボックスから下のテキスト ボックスへの一方向だけで、逆方向は行えません。そこで両方向に行えるように改造してみます。このようにするには、コントロール配列を使う方法や、**SCREEN** オブジェクトの ActiveControl プロパティと **IF TYPEOF** ステートメントを使う方法な

ど、いくつかのパターンが考えられますが、ここでは最も簡単な Tag プロパティと **IF** ステートメントを使う方法を考えてみます。Tag プロパティはフォームやコントロールに対するコメントとして利用できるプロパティで、オブジェクトの外観や動作には影響しません。

カット & ペーストの機能をコマンド ボタンを使って実現するとき、ユーザーがボタンを押したときにはフォーカスはテキスト ボックスではなくコマンド ボタンにあるため、ユーザーがどちらのテキスト ボックスに対して操作したいのかを判断するためには、何らかの別の情報が必要です。ここではテキスト ボックスからコマンド ボタンにフォーカスが移るとき、テキスト ボックスの LostFocus イベントに対するイベント プロシージャを使って、フォームの Tag プロパティに、どちらのテキスト ボックスが最後に操作されていたのかを示す文字列を代入しておくようにします。

▼ 両方向のカット & ペーストができるように改造するには

1. 両方のテキスト ボックスに対して、LostFocus イベントに対応する次のようなイベント プロシージャを追加します。

```
SUB Text1_LostFocus ()
    Form1.Tag = "Text1"
END SUB

SUB Text2_LostFocus ()
    Form1.Tag = "Text2"
END SUB
```

2. 前に作成した 3 つのイベント プロシージャを、次のように変更します。

```
SUB cmdCut_Click ()
    IF Form1.Tag = "Text1" THEN
        CLIPBOARD.SETTEXT Text1.SelText
        Text1.SelText = ""
    ELSE
        CLIPBOARD.SETTEXT Text2.SelText
        Text2.SelText = ""
    END IF
END SUB

SUB cmdCopy_Click ()
    IF Form1.Tag = "Text1" THEN
        CLIPBOARD.SETTEXT Text1.SelText
        Text1.SelStart = Text1.SelStart + Text1.SelLength
    ELSE
        CLIPBOARD.SETTEXT Text2.SelText
        Text2.SelStart = Text2.SelStart + Text2.SelLength
    END IF
END SUB

SUB cmdPaste_Click ()
    IF Form1.Tag = "Text1" THEN
        Text1.SelText = CLIPBOARD.GETTEXT()
    ELSE
        Text2.SelText = CLIPBOARD.GETTEXT()
    END IF
END SUB
```

cmdCopy_Click イベント プロシージャに追加されている行 (4 行目と 7 行目) は、テキスト ボックスから別のコントロールにフォーカスが移ったとき、選択されているテキストが反転表示されなくなるため、ユーザーが混乱しないように選択を解除しています。

クリップボードのクリア

CLEAR メソッドは、そのとき **CLIPBOARD** オブジェクトに格納されている内容をクリアします。**CLEAR** メソッドの構文を次に示します。

CLIPBOARD.CLEAR

7.3　フォームおよびピクチャー ボックスへの出力

テキストは、テキスト ボックスに表示するだけでなく、フォームやピクチャー ボックスに直接出力することもできます。ただし、これらのテキストをユーザーが編集することはできません。

注　意　フォームまたはピクチャー ボックスに表示しきれない長さの文字列を出力すると、右または下にはみ出した部分は無視され、表示されません。フォームまたはピクチャー ボックスはスクロールしません。

フォームやピクチャー ボックスにテキストを表示するには、**PRINT** メソッドを使います。**PRINT** メソッドは **PRINT** ステートメントに似ていますが、**PRINT** ステートメントがデスクトップ (画面) に直接出力するのに対し、**PRINT** メソッドはフォーム，ピクチャー ボックス、および **PRINTER** オブジェクトに対してだけ有効です。

フォーム モジュールに含まれる **PRINT** は常に **PRINT** メソッドとして扱われます。デスクトップに文字を出力するには、コード モジュールで **PRINT** ステートメントを実行する必要があります。ただし、フォームが表示されている状態では **PRINT** ステートメントは実行できないので注意してください。次に **PRINT** ステートメントの実行例を示します。

```
'FPRINT.BAS
SUB Fprint (X AS STRING)
    SCREEN.HIDE
    PRINT X
END SUB
```

また、**CLS** メソッドと **CLS** ステートメントにも同じことが当てはまります。

PRINT メソッドの構文を次に示します。

[*object*.] **PRINT** [*expressionlist*] [{; | ,}]

引数	説明
object	出力するフォームまたはピクチャー ボックスの名前，または **PRINTER** オブジェクト (**PRINTER** オブジェクトについては、この章の「プリンタへのテキストの出力」を参照してください)。
expressionlist	出力するデータのリスト

object は省略できます。*object* を省略すると、**PRINT** メソッドはそのときアクティブなフォームに出力します。

注 意 Form_Load イベント プロシージャで **PRINT** メソッドを使う場合は、文字を出力するフォームまたはピクチャー ボックスの AutoRedraw プロパティを True (-1) に設定してください。フォームやピクチャー ボックス内のテキストは**SHOW** メソッドを実行すると消去されてしまうため、AutoRedraw プロパティがデフォルトの設定である False (0) のままでは、フォームが表示されたときにテキストが表示されません。Paint イベントに対するイベント プロシージャで **PRINT** メソッドを使用する場合は、AutoRedraw プロパティの設定にかかわらず常にテキストが表示されます。

例えば次のステートメントは、MyForm，Picture1，現在のフォームにそれぞれ異なるメッセージを出力します。

```
MyForm.Print "これはフォーム MyForm に出力されます"
Picture1.Print "これはピクチャー ボックスに出力されます"
Print "これはそのときアクティブなフォームに出力されます"
```

コード モジュールで **PRINT** メソッドを使うときは、出力対象のオブジェクトの指定を省略できません。次に例を示します。

```
MyForm.Picture1.Print "これはピクチャー ボックスに出力されます"
```

データの区切りと出力位置

数値データを出力すると、その後ろには自動的にスペースが 1 つ入れられます。また、数値が正の値なら、数値の前にもスペースが 1 つ挿入されます。数値が負の値なら、数値の前にマイナス記号 (-) が付けられます。文字列データに対してはスペースは挿入されません。

1 つの **PRINT** メソッドで複数のデータを出力するときは、セミコロン (;) またはカンマ (,) で区切ります。どちらを指定するかによって、次のデータが出力される位置が異なります。また、最後のデータの後にセミコロンまたはカンマを付けると、次の **PRINT** メソッドで出力するデータの位置を制御できます。

- データの後にセミコロンを付けると、そのデータを出力した後、出力位置は進められません。次のデータは、その直後から出力されます。
- データの後にカンマを付けると、そのデータを出力した後、出力位置は次の出力領域の先頭に進められます。出力領域は、行の左端から 14 桁ごとに区切られています。

- 最後のデータの後にセミコロンまたはカンマを指定しないと、次のデータは次の行の先頭から出力されます。

例えば次のステートメントは、現在のフォームに対して出力します。

```
PRINT "X の値は"; X; "で Y の値は"; Y
```

X の値が 2 で Y の値が -7 のときは、次のように出力されます。

```
X の値は 2 で Y の値は-7
```

次のように指定すると、ピクチャー ボックス Picture1 に 2 行のテキストが表示されます。

```
Picture1.PRINT "これが 1 行目"
Picture1.PRINT "これが 2 行目"
```

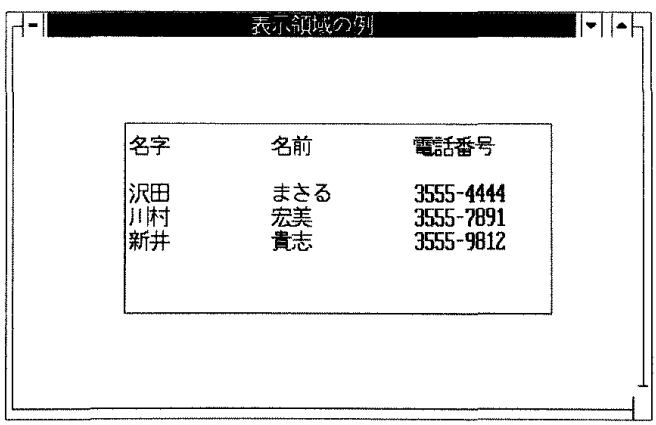
最初のステートメントの最後にセミコロンを付けると、出力は 1 行になります。

```
Picture1.PRINT "これらすべてが";
Picture1.PRINT "同じ行に表示される"
```

フォームやピクチャー ボックス，デスクトップなどは、14 桁ごとの表示領域に区切られています。**PRINT** メソッドのデータの区切りとしてカンマを指定すると、この表示領域に従ってテキストを表示することができます。

```
Picture1.Print "名字", "名前", "電話番号"
Picture1.Print
Picture1.Print "沢田", "まさる", "3555-4444"
Picture1.Print "川村", "宏美", "3555-7891"
Picture1.Print "新井", "貴志", "3555-8912"
```

実行すると、3 つの表示領域に出力されます。



桁位置の設定

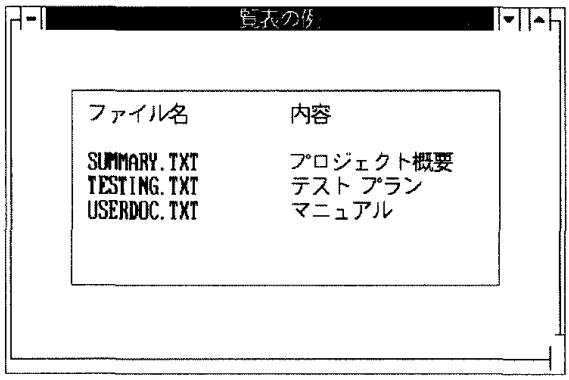
PRINT メソッド中で **TAB** 関数を使うと、次のデータを出力する桁位置を指定できます。**TAB** 関数の構文を次に示します。

TAB (*column*) ;

例えば次のコードを実行すると、メッセージは 5 列目から表示されます。

```
Picture1.Print Tab(5); "このメッセージは 5 桁目から出力されます"
```

次の図は、ピクチャー ボックスに出力されたファイル名の一覧です。**TAB** 関数を使って、2 桁目と 22 桁目に合わせて出力しています。



この一覧表を出力するコードを次に示します。

```
Picture1.Print Tab(2); "ファイル名"; Tab(22); "内容"
Picture1.Print
Picture1.Print Tab(2); "SUMMARY.TXT"; Tab(22); "プロジェクト概要"
Picture1.Print Tab(2); "TESTING.TXT"; Tab(22); "テスト プラン"
Picture1.Print Tab(2); "USERDOC.TXT"; Tab(22); "マニュアル"
```

出力位置の指定

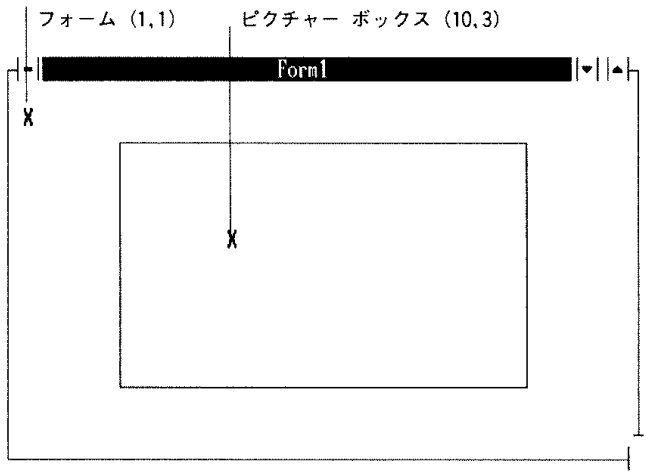
フォームまたはピクチャー ボックスの CurrentX および CurrentY プロパティは、**PRINT** メソッドで次に出力される位置を示しています。このプロパティの値を変更すると、出力する位置を制御できます。CurrentX および CurrentY プロパティが保持している座標は、フォームまたはコントロールの左上角 (0,0) に対する相対位置です。0 ～ 255 の値を指定できます。CurrentX および CurrentY プロパティの値はコード中から取得および設定できますが、デザイン時にフォーム デザイナで設定することはできません。

注　意　出力する位置にすでに他のテキストが表示されているときは上書きされます。

例えば次のステートメントは、フォーム上の出力位置を (1,1) に、ピクチャー ボックス上の出力位置を (10,3) に設定します。

```
CurrentX = 1            'フォーム上の次の出力位置を設定します.
CurrentY = 1
Picture1.CurrentX = 10        'ピクチャー ボックス上の次の出力位置を
Picture1.CurrentY = 3         '設定します.
```

次に **PRINT** メソッドでテキストを出力すると、次の図の位置から表示されます。



CLS メソッドは、前の **PRINT** メソッドでオブジェクトに出力されたすべてのテキストを消去し、CurrentX および CurrentY プロパティを左上角 (0,0) にリセットします。例えば次のステートメントは、Picture1 と現在のフォームの内容を消去します。

```
Picture1.CLS            'ピクチャー ボックス上のテキストをクリアします.
CLS                     'フォーム上のテキストをクリアします.
```

テキストの高さと幅

テキスト文字列の高さ (行数) や幅 (文字数) を調べるときは、**TEXTHEIGHT** メソッドと **TEXTWIDTH** メソッドを使います。**TEXTHEIGHT** および **TEXTWIDTH** メソッドの構文を次に示します。関数のように式の中で使うことに注意してください。

[*object.*] [**TEXTHEIGHT** | **TEXTWIDTH**] (*string*)

引数	説明
object	フォームまたはピクチャー ボックスの名前，または **PRINTER** オブジェクト。省略するとそのときアクティブなフォームと見なされます。
string	高さまたは長さを調べる文字列

指定した文字列にキャリッジリターン (CHR$(13)) やラインフィード (CHR$(10)) が埋め込まれているときは、複数行に展開され、**TEXTHEIGHT** メソッドはその行数を返します。埋め込まれていなければ、長さにかかわらず 1 を返します。

TEXTWIDTH メソッドは文字列の改行位置を調整するときに便利です。表示するテキストがフォームやピクチャー ボックスの右端からはみ出す可能性のあるときは、**TEXTWIDTH** メソッドで調べることができます。

注 意 ScaleWidth および ScaleHeight プロパティは、オブジェクトの内側のサイズ、つまりオブジェクトの境界内の領域の高さと幅を示します。フォームとピクチャー ボックスだけがこれらのプロパティを持っています。

7.4 数値，日付，時刻の書式の指定

数値，日付，時刻を出力するときの書式を指定するには、**FORMAT$** 関数を使います。**FORMAT$** 関数には、出力するデータと、その出力書式を示す文字列の 2 つの引数を指定します。結果として、書式に従って変換された文字列が返されます。

FORMAT$ 関数の構文を次に示します。

FORMAT$ (*expression* [, *fmt$*])

引数	説明
expression	変換するデータを示す数式
fmt$	数値や日付，時刻の書式を指定する文字を含む文字列式

数値データの書式の指定

FORMAT$** 関数は **STR$ 関数と同じように、数値を文字列に変換します。小数点以下の桁数，前後の 0，通貨書式などを指定できるので、さらに細かく文字列の表示を制御できます。書式を指定するには、引数 *fmt$* に、表示書式を示す文字を組み合わせた文字列を指定します。よく使われる書式指定文字には、次のようなものがあります。詳しくは、『リファレンス』またはオンライン ヘルプを参照してください。

シンボル	説明
0	1 つを 1 桁として、桁位置と桁数を指定します。これらの桁は必ず埋められ、出力する数字がない桁には 0 が出力されます。
#	0 と同じ機能ですが、出力する数字がない桁には何も出力されません。
.	小数点の位置を指定します。
,	小数点以上 3 桁ごとのカンマを指定します。

次にいくつかの例を示します。

FORMAT$ の指定	結果
FORMAT$(8315.4, "00000.00")	08315.40
FORMAT$(8315.4, "#####.##")	8315.4
FORMAT$(8315.4, "##,##0.00")	8,315.40
FORMAT$(315.4 ,"¥##0")	¥540

日付と時刻の書式の指定

日付と時刻の書式を出力するには、対応する書式指定文字を使います。次の例では、**NOW** 関数を使って現在の日付または時刻を取得し、書式を指定して出力しています。

FORMAT$ の指定	結果
FORMAT$(NOW, "yyyy年m月d日")	1998年1月31日
FORMAT$(NOW, "yyyy/mm/dd ")	1998/01/31
FORMAT$(NOW, "yy/m/d")	98/12//7
FORMAT$(NOW, "yy-mmmm-d")	98-December-7
FORMAT$(NOW, "h:mm AM/PM")	8:50 PM
FORMAT$(NOW, "h:mm:ss am/pm")	8:50:35 pm
FORMAT$(NOW, "yy/m/d h:mm")	98/12/7 20:50

7.5　プリンタへの出力

Visual Basic では、データを直接プリンタに出力することもできれば、画面に表示されているフォームのイメージをプリンタに出力することもできます。特殊オブジェクトの 1 つである **PRINTER** オブジェクトが、データの出力先であるプリンタまたはファイルを示し、出力を制御します。

出力先の指定

出力するプリンタを指定するには、**PRINTER** オブジェクトの PrintTarget プロパティを使います。例えば LPT1 に出力するときは、PrintTarget プロパティを次のように設定してください。必ずコロン (:) を付けることに注意してください。

```
PRINTER.PrintTarget = "LPT1:"
```

ファイル名を指定すると、出力先をファイルに設定することもできます。

プリンタへのテキストの出力

テキストを直接プリンタに出力するには **PRINTER** オブジェクトに対する **PRINT** メソッドを使います。

PRINTER.PRINT *expressionlist* [{; | ,}]

長い文書を出力するときは、**NEWPAGE** メソッドを使って改ページする位置を指定したり、**ENDDOC** メソッドを使って文書の最後を指定できます。ページ プリンタやネットワークを介して接続されているプリンタなど出力の最後の位置が意味を持つ場合は、出力の最後に必ず **ENDDOC** メソッドを使うようにしてください。例えば次のステートメントを実行すると、1 ページ目を出力して改ページし、2 ページ目を出力して終了します。

```
PRINTER.PRINT "1 ページ目"
PRINTER.NEWPAGE                '改ページします.
PRINTER.PRINT "2 ページ目"
PRINTER.ENDDOC                 '出力を終了します.
```

フォームの出力

画面に表示されているフォームのイメージをプリンタに出力することもできます。この場合、キャラクタを組み合わせて疑似的にイメージを再現します。フォームのイメージをプリンタまたはファイルに出力するには、**PRINTFORM** メソッドを使います。**PRINTFORM** メソッドはフォームの一部が画面に表示されていないときでも、フォーム全体を出力します。

PRINTFORM の構文を次に示します。

[*form.*] **PRINTFORM**

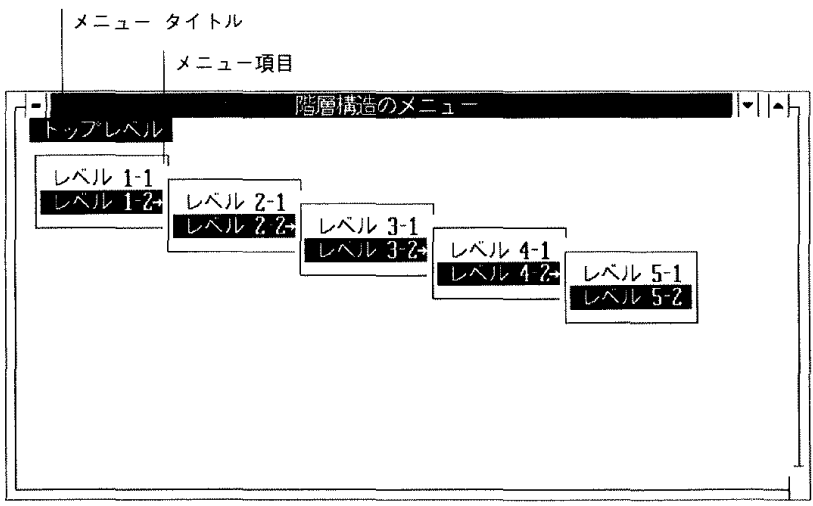
フォーム名を省略すると、そのときアクティブになっているフォームが出力されます。次に例を示します。

```
PRINT "フォームに表示されるテキスト"
                                'AutoRedraw は True に設定します.
MyForm.PRINTFORM
PRINTER.ENDDOC
```

8 メニューの作成

Visual Basic には、アプリケーションのメニューを作成するための機能が用意されています。この機能を利用すると、プログラミング環境やフォーム デザイナで使っているものと同じような本格的なメニューを、作成するアプリケーションに簡単に追加できます。メニューを活用することによって、アプリケーションの操作性を向上させることができます。

プログラミング環境での [ﾌｧｲﾙ], [表示] など、メニューの内容を示しているのがメニュー タイトルです。画面の一番上の、メニュー タイトルが並んでいる行を、メニュー バーと呼びます。メニュー タイトルをクリックしたときにメニューがオープンする動作を、“メニューがドロップダウンする”と言います。ドロップダウンしたメニューには、メニュー コマンドの一覧が含まれています。それぞれのメニュー コマンドを、“メニュー項目”と呼びます。また、メニュー項目をグループ分けするために引かれている横線が、セパレータ バーです。Visual Basic では、5 レベルまでメニューを階層化できます。



各メニュー タイトル，メニュー項目，セパレータ バーは、それぞれ 1 つのメニュー コントロールです。ユーザーがメニュー タイトルやメニュー項目を選択すると、そのメニュー コントロールに対する Click イベントが発生します。最上位のメニュー コントロールであるメニュー タイトルは、メニュー項目の集まりであるメニューを持っています。また、それぞれのレベルのメニュー項目に対しても、サブメニューを設定できます。このようにメニューやサブメニューを持つメニュー コントロールをユーザーがクリックしたときは、メニューやサブメニューは自動的にドロップダウンしますので、

特にプログラミングは必要ありません。通常は、これらのメニューを持つものを除くメニュー コントロールに対して、Click イベント プロシージャを記述して行くことによって、メニュー インターフェイスを作成して行きます。

この章では、アプリケーションのメニュー インターフェイスを設計する方法と、次の項目について説明します。

- アクセス キーとショートカット キーの設定
- セパレータ バーの挿入
- メニュー コントロールの無効化
- メニュー項目へのチェック マークの追加
- メニュー コントロールを非表示にする方法
- 階層構造を持つメニュー
- 実行時のメニュー コントロールの追加と削除
- マルチ ドキュメント インターフェイス (MDI) アプリケーションの作成。MDI アプリケーションは、1 つのメニュー インターフェイスで複数のウィンドウを制御します。

8.1　メニュー インターフェイスの作成

メニューは、さまざまな機能を呼び出すメニュー コントロールの集まりです。他のコントロールと異なり、フォームへの配置やプロパティの設定には、ツールボックスやプロパティ バーではなく、メニュー デザイン ウィンドウを使います。

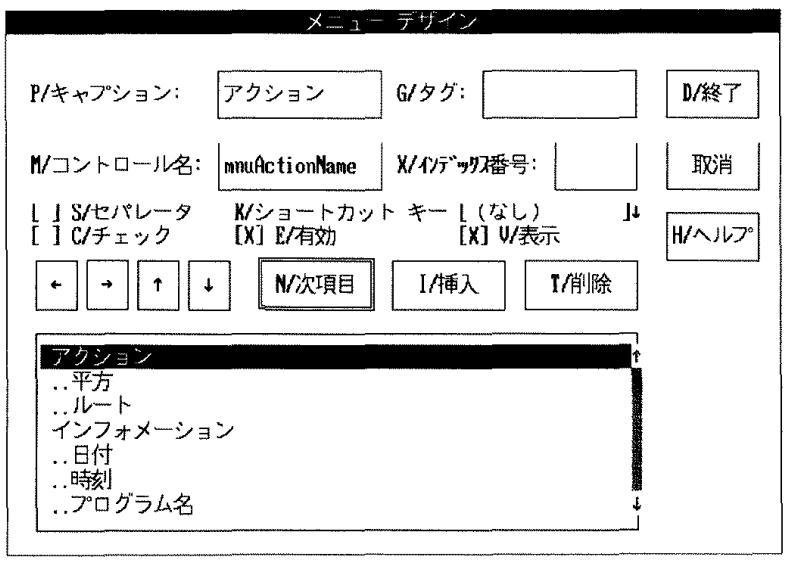


図 8.1　メニュー デザイン ウィンドウ

このメニュー デザイン ウィンドウ上で、メニュー コントロールを追加し、各コントロールのプロパティを設定することによって、メニューを作成して行きます。メニュー デザイン ウィンドウの一番下のリスト ボックスには、設定されているメニュー コントロールの一覧が表示されます。各メニューがどのレベルに属しているかは、インデントで示されます。リストの中で反転表示されているコントロールに対する設定が、メニュー デザイン ウィンドウの上半分に表示されます。

メニュー デザイン ウィンドウでメニューの外観とメニュー コントロールのプロパティを設定した後、プログラミング環境で各メニュー コントロールの Click イベント プロシージャを記述します。

メニュー デザイン ウィンドウでは、メニュー コントロールのプロパティが次のように表記されています。

Caption	キャプション
CltName	コントロール名
Tag	タグ
Index	インデックス番号
Separator	セパレータ
Checked	チェック
Enabled	有効
Visible	表示

▼ メニュー デザイン ウィンドウでメニュー コントロールを作成するには

1. メニューを作成するフォームを指定し、フォーム デザイナに移ります。
2. [ｳｨﾝﾄﾞｳ] メニューの [メニュー デザイン] コマンドを選択します。

   メニュー デザイン ウィンドウが表示されます。
3. メニュー コントロールに対して表示したいテキストを、[キャプション] テキスト ボックスに入力します。

   このテキストは、自動的に Caption プロパティに設定され、メニュー デザイン ウィンドウのリスト ボックスに表示されます。
4. [コントロール名] テキスト ボックスに、そのコントロールの名前 (CtlName プロパティの値) を入力します。
5. メニュー項目を追加するときは、<次項目> ボタンをクリックします。また、既存のメニュー項目の間に新たなメニュー項目を挿入するときは、<挿入> ボタンをクリックします。

   <→> ボタンと <←> ボタンは、そのとき反転表示されている項目の階層を設定するために使います。階層は、インデントされてリスト ボックス表示されることによって示されます。リスト ボックスに連続して表示されている、同じ階層のメニュー項目が、1 つのメニューを構成することになります。

   <↑> ボタンと <↓> ボタンは、メニュー コントロールの順番を入れ換えたいときに使います。

6. 必要なメニュー コントロールをすべて作成するまで、ステップ 3 ～ 5 を繰り返します。
7. メニュー コントロールの作成を終了するときは、<終了> ボタンをクリックします。

メニュー デザイン ウィンドウをクローズすると、作成したメニューがフォーム上に表示されます。メニュー デザイン ウィンドウでは、Caption および CtlName プロパティ以外にも、Tag, Index, Checked, Visible, Enabled, Separator の各プロパティを設定できます (詳しくは、「メニュー コントロールのプロパティの設定」を参照してください)。

▼ **既存のメニューを編集するには**

1. メニューを作成するフォームを指定し、フォーム デザイナに移ります。
2. [ｳｨﾝﾄﾞｳ] メニュー の [メニュー デザイン] コマンドを選択します。
3. 編集するメニュー コントロールをリスト ボックスの中から選択し、必要な変更を行います。
4. 終了するときは <終了> ボタンをクリックします。

メニュー コントロールのプロパティの設定

メニュー デザイン ウィンドウでは、デザイン時に設定できるすべてのプロパティがダイアログ ボックスに表示されます。プロパティを設定するときは、まずリスト ボックスの中からメニュー コントロールを選択します。

- Caption (キャプション), CtlName (コントロール名), Tag (タグ), Index (インデックス番号) の各プロパティを設定するときは、テキスト ボックスに値を入力します。
- Checked (チェック), Visible (表示), Enabled (有効), Separator (セパレータ) の各プロパティを設定するときは、対応するチェック ボックスをチェックします。

それぞれのプロパティの詳細については、オンライン ヘルプまたは『リファレンス』を参照してください。

注　意　メニュー コントロールの Parent プロパティは、実行時のみ設定できます。詳しくはオンライン ヘルプを参照してください。

8.2　メニュー コントロールに対応するコードの記述

メニュー コントロールによって呼び出される Click イベント プロシージャは、他のコントロールによって呼び出されるイベント プロシージャと同じように作成します。この方法の詳細については、第 2 章「Hello, world アプリケーションの作成」を参照してください。

メニュー タイトルやメニュー項目はそれぞれが独立したコントロールであり、ユーザーがその項目を選択すると Click イベントが発生します。セパレータ バー以外のメニュー

コントロール (Separator プロパティが 0 に設定されているメニュー コントロール) はすべて Click イベントを認識します。セパレータ バーに対してもイベント プロシージャを記述することはできますが、イベントが発生しないため、実行されることはありません。

一般に、メニュー タイトルや、サブメニューを持つメニュー項目にはイベント プロシージャは記述しません。あるメニュー項目がサブメニューを持っている場合、そのメニュー項目は下のレベルのメニュー項目を表示するためのメニュータイトルであると考えることができます。この場合、上位のメニュー項目を選択すると、自動的にサブメニューが表示されるため、プログラマが Click イベント プロシージャを記述する必要はありません。

多くの場合、メニュー項目がクリックされたとき、ダイアログ ボックスを表示してユーザーに対してより詳しい入力を求める必要があります。例えば [ファイル オープン] コマンドでは、オープンするファイル名を指定する必要があります。このようなメニュー項目に対しては、Click イベント プロシージャにダイアログ ボックスを表示するための必要な処理を記述してください (ダイアログ ボックスについては、第 5 章「ダイアログ ボックスの作成」を参照してください)。

注 意 ダイアログ ボックスの表示を行うメニュー項目のキャプションには、最後に省略記号 (...) を付けることをおすすめします。多くのアプリケーションでは、省略記号が付いているメニュー コマンド (メニュー項目) は、選択するとダイアログ ボックスが表示されることを示しています。このようなルールに従うことによって、ユーザーがより理解しやすいユーザー インターフェイスを作成できます。

8.3 Try Menus アプリケーションの作成

ここでは、Try Menus という名前のアプリケーションを使って、メニューの作成方法とその使い方を説明します。Try Menus アプリケーションのフォームには、[アクション] と [インフォメーション] の 2 つのメニューが含まれます。フォームの外観を図 8.2 に示します。左が [アクション] メニューで、右が [インフォメーション] メニューです。

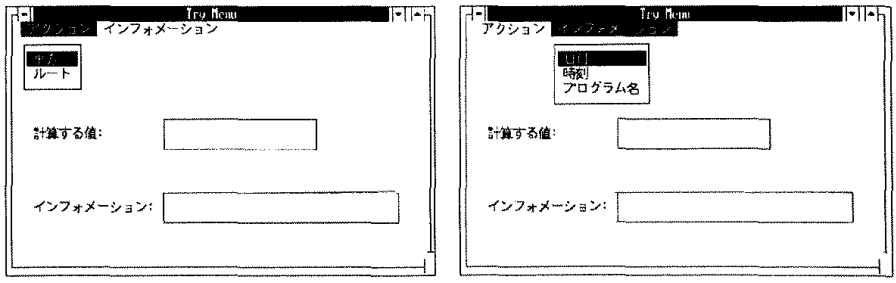


図 8.2 [アクション] メニューと [インフォメーション] メニュー

Try Menus アプリケーションのメニューは次のように動作します。

- [アクション] メニューの [平方] コマンドと [ルート] コマンドは、テキスト ボックスの最初に表示されている数値に対して計算を行います。

- [インフォメーション] メニューの [日付], [時刻], [プログラム名] コマンドを選択すると、テキスト ボックスの一番下に情報を表示します。

2 つのメニューに含まれる各コマンドには、対応する Click イベント プロシージャを記述します。メニュー タイトル ([アクション] と [インフォメーション]) に対しては、イベント プロシージャを記述する必要はありません。

キャプション (Caption)	コントロール名 (CtlName)	インデント
アクション	mnuActionName	インデントしない
平方	mnuSquareCom	レベル 1
ルート	mnuSquareRtCom	レベル 1
インフォメーション	mnuInfoName	インデントしない
日付	mnuDateCom	レベル 1
時刻	mnuTimeCom	レベル 1
インフォメーション	mnuInfo	レベル 1

▼ Try Menus アプリケーションを作成するには

1. メニュー デザイン ウィンドウで、メニュー コントロールのプロパティを設定します。まず、キャプション (Caption) テキスト ボックスに次のように入力します。

   アクション

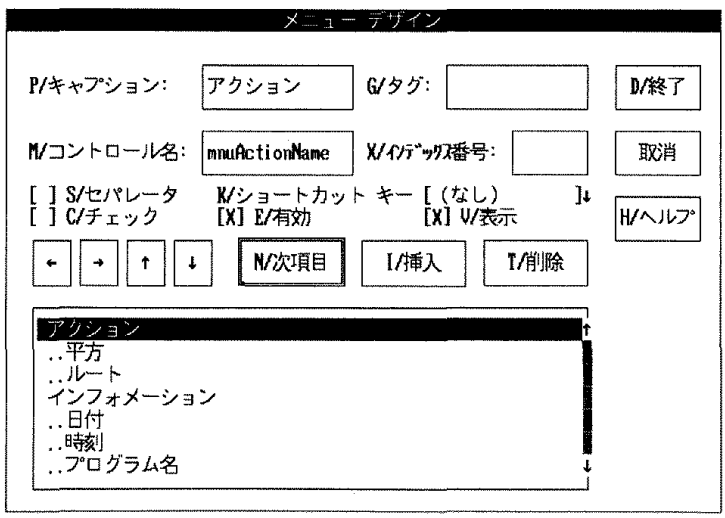
   [TAB] キーを押してコントロール名 (CtlName) テキスト ボックスにフォーカスを移します。リスト ボックスに “アクション” と表示されます。

2. コントロール名テキスト ボックスに、次のように入力します。

   `mnuActionName`

3. <次項目> ボタンをクリックします。

4. 以下、次の図のようになるまで、設定を続けます。



メニュー コントロールの一覧が表示されているリスト ボックス内のインデントに注意してください。このメニューは階層化されていて、メニュー タイトルである“アクション”と“インフォメーション”の下に、それぞれいくつかのメニュー項目が配置されています。メニュー タイトルはインデントされていませんが、その他のメニュー項目はインデントされています。

メニュー項目を階層化するには、メニュー デザイン ウィンドウの左側にある <→> をクリックします。次の項目も同じようにインデントされますが、<←> と <→> ボタンを使って変更できます。

5. メニューの設定が終了したら、<終了> ボタンを押して、メニュー デザイン ウィンドウを閉じます。

6. 2 つのラベルと 2 つのテキスト ボックスをフォームに追加し、次の表に従ってコントロールのプロパティを設定します。終了したら、プログラミング環境に戻ります。

コントロール	プロパティの設定
フォーム	Caption = Try Menus
上のテキスト ボックス	CtlName = Readout Text= (なし)
下のテキスト ボックス	CtlName = InfoBox Text= (なし)
上のラベル	CtlName = Label1 Text = 計算する値
下のラベル	CtlName = Label2 Text = インフォメーション

7. メニュー項目 [平方] の Click イベント プロシージャに次のコードを追加します。

```
SUB mnuSquareCom_Click ()
    Amt = VAL(Readout.Text)
    Amt = Amt * Amt
    Readout.Text = STR$(Amt)
END SUB
```

このプロシージャでは、テキスト ボックスに入力されたテキストを数値に変換し、平方を計算してから、結果を書き戻しています。

8. メニュー項目[ルート] の Click イベント プロシージャに次のコードを追加します。

```
SUB mnuSquareRtCom_Click ()
    Amt = VAL(Readout.Text)
    Amt = SQR(Amt)
    Readout.Text = STR$(Amt)
END SUB
```

このプロシージャでは、**SQR** 関数を使って平方根を求めています。

9. メニュー項目 [日付] の Click イベント プロシージャに次のコードを追加します。

```
SUB mnuDateCom_Click ()
    InfoBox.Text = DATE$
END SUB
```

このプロシージャは、**DATE$** 関数を使って、InfoBox テキスト ボックスに現在の日付を表示します。

10. メニュー項目 [時刻] の Click イベント プロシージャに次のコードを追加します。

```
SUB mnuTimeCom_Click ()
    InfoBox.Text = TIME$
                        'システムの時間をテキスト ボックスに表示します.
END SUB
```

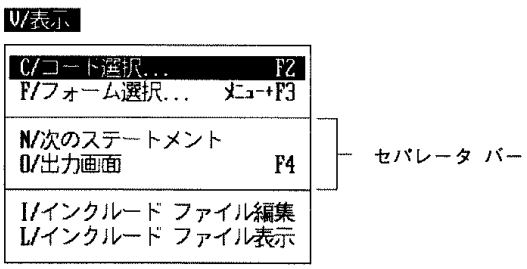
同様に **TIME$** 関数を使って時刻を表示します。

11. メニュー項目 [プログラム名] の Click イベント プロシージャに次のコードを追加します。

```
SUB mnuProgNameCom_Click ()
    InfoBox.Text = "プログラム名 is Try Menus."
END SUB
```

8.4　セパレータ バーの使い方

“セパレータ バー” はメニュー上でメニュー項目を区切る横棒です。セパレータ バーもメニュー コントロールの 1 つですが、Click イベントは発生せず、タブ オーダーにも含まれません。アプリケーションのユーザーはセパレータ バーを選択することはできません。1 つのメニューに多くのメニュー項目が含まれるときは、セパレータ バーを使うと、メニュー項目をいくつかのグループに分け、見やすくすることができます。



▼ **セパレータ バーを作成するには**

1. フォーム デザイナでメニュー デザイン ウィンドウをオープンします。
2. リスト ボックス中で、セパレータ バーを挿入したい位置を反転表示し、<挿入> ボタンをクリックします。
3. 前後のメニュー項目と同じレベルまでインデントします。
4. [コントロール名] リスト ボックスにコントロール名を入力し、CtlName プロパティを設定します。

5. [セパレータ] チェック ボックスをチェックし、Separator プロパティを True (-1) に設定します。

6. <終了> を選択します。

セパレータ バーも 1 つのメニュー コントロールですが、実際に表示されるのは横棒だけです。Caption プロパティは設定できますが意味を持ちません。つまり、Separator プロパティが True (-1) に設定されていると、Caption プロパティは無視されます。

8.5 アクセス キーとショートカット キーの設定

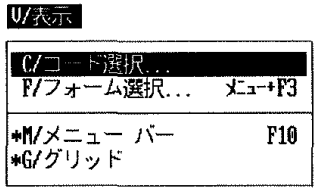
各メニュー項目には、アクセス キーとショートカット キーを設定できます。またメニュー タイトルには、アクセス キーだけが設定できます。

■ アクセス キー

[F/ﾌｧｲﾙ] メニューの [ﾒﾆｭｰ]+[F] のように、特定のアルファベットや数字キーを押すことによって、メニューをドロップダウンさせたり、メニュー項目を選択できるようにするキーです。通常、アクセス キーは [ﾒﾆｭｰ] キーと同時に押しますが、メニューからメニュー項目を選択するときは [ﾒﾆｭｰ] キーを押す必要はありません。アクセス キーに割り当てられている文字は、**SCREEN** オブジェクトの ControlPanel プロパティの設定に従って、他の文字と別の色で表示されます。2 バイト文字やカタカナをアクセス キーに設定することはできません。通常は、“F/ファイル” のように先頭にアルファベットを付け、その文字をアクセス キーにします。また、Microsoft Windows のように、メニュー タイトルの後ろにカッコで囲んでアクセス キーを表示するのもいい方法です。

■ ショートカット キー

よく使うメニュー項目に割り当てる、メニューをドロップダウンせずに、そのメニュー コマンドを実行するためのキーです。通常、ファンクション キーや、コントロール キーと文字の組み合わせ ([CTRL]+[A] など) を割り当てます。メニュー上では、設定したショートカット キーはメニュー項目の右側に表示されます。



▼ アクセス キーを設定するには

1. フォーム デザイナでメニュー デザイン ウィンドウをオープンします。

2. アクセス キーを設定したいメニュー項目を選択します。

3. [キャプション] テキスト ボックスで、アクセス キーとして設定する文字の前にアンパサンド (&) を入力します。

Caption プロパティの設定例と対応するアクセス キーを次に示します。

Caption プロパティの設定	アクセス キー
&O/オープン	[メニュー] + [O]
クローズ (&C)	[メニュー] + [C]
&X/終了	[メニュー] + [X]

注　意　1 つのメニューの中で、同じアクセス キーを複数のメニュー項目に割り当てないように注意してください。

▼ **ショートカット キーを割り当てるには**

1. フォーム デザイナでメニュー デザイン ウィンドウをオープンします。
2. ショートカット キーを設定したいメニュー項目を選択します。
3. [ショートカット キー] を選びます。これはドロップ ダウン リスト ボックスになっており、右側の下矢印をクリックすると、設定可能なキーの組み合わせをすべて含むリスト ボックスがドロップダウンします。

   ショートカット キーなしに設定したいときは、リストの先頭にある (なし) を選択してください。なしがデフォルトです。

8.6　メニュー コントロールを無効にする

状況によって、特定のメニュー項目をユーザーが選択できないようにしたいことがあります。このような場合、メニュー コントロールの Enabled プロパティを False (0) に設定すると、そのメニュー コントロールは無効になり、ユーザーがクリックしてもイベントが発生しないようになります。無効になっているメニュー コントロールは、**SCREEN** オブジェクトの ControlPanel プロパティの設定に従って、有効なコントロールとは別の色で表示されます。

デザイン時にメニュー コントロールの有効/無効を切り換えたいときは、メニュー デザイン ウィンドウで [有効] チェック ボックスを操作します。実行時に切り換えたいときは、Enabled プロパティに True (-1) または False (0) を代入します。

```
mnuFileSave.Enabled = -1          '[ファイル保管]コマンドを有効にする
mnuFileSave.Enabled = 0           '[ファイル保管]コマンドを無効にする
```

サブメニューを持つメニュー項目やメニュー タイトルを無効にすると、その下位のメニューは選択できなくなります。

空白のメニュー タイトル

Enabled プロパティを False (0) に設定し、Caption プロパティにスペースを設定すると、空白のメニュー タイトルやメニュー項目を作成できます。通常、メニューを使うアプリケーションでは、タイトル バーの右端に [ヘルプ] メニューがあり、その左側が

少し空いています。このような効果を出したいときに、このテクニックを使います。Visible プロパティが True (-1) に設定されていることを確認してください。

8.7 チェック マークの表示

オン/オフを切り換えるメニュー コマンドでは、チェック マークを使ってその状態を示すことができます。チェック マークは、メニュー項目の左側に表示されるアスタリスク (*) です。

デザイン時にチェック マークのあり/なしを切り換えたいときは、メニュー デザイン ウィンドウで [チェック] チェック ボックスを操作します。実行時に切り換えたいときは、Checked プロパティに True (-1) または False (0) を代入します。次の例では、メニュー項目が選択されるとチェック マークのあり/なしが切り換えられます。

```
SUB mnuFullMenus_Click
    mnuFullMenus.Checked = NOT mnuFullMenus.Checked
END SUB
```

8.8 メニュー コントロールを非表示にする

メニュー コントロールは、一時的に非表示にすることができます。非表示にすると、その下に続くメニュー コントロールが順に繰り上げられます。メニュー バー上のメニュー タイトルを非表示にすると、残りのメニュー タイトルは左に移動します。

デザイン時に表示/非表示を切り換えたいときは、メニュー デザイン ウィンドウで [表示] チェック ボックスを操作します。実行時に切り換えたいときは、Visible プロパティに True (-1) または False (0) を代入します。

```
mnuStockQuote.Visible = -1        'コントロールを表示状態にします.
mnuStockQuote.Visible = 0         'コントロールを非表示にします.
```

注 意 メニュー コントロールを非表示にすると、そのコントロールは自動的に無効になります。したがって、メニュー タイトルを非表示にすると、そのメニュー タイトルの下位の階層に含まれるすべてのメニューが無効になります。

8.9 メニューの階層化

Visual Basic では、あるメニュー項目を選択したときに、さらに別のメニューが表示されるように設定することができます。この機能を使うと、5 レベルまでメニューを階層化できます。

2 レベルのメニューを持つ例を次に示します。サブメニューを持つメニュー項目には、名前 (キャプション) の右端に右向きの矢印が表示されます。

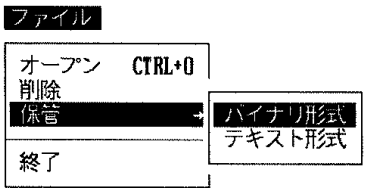


図 8.3　2 レベルのサブメニューを含むメニュー

メニュー タイトルの数が少なく、メニュー バーの横幅に余裕があるときは、メニューを階層化せずに、新しくメニュー タイトルを作成することをおすすめします。メニューをあまり深く階層化してしまうと、アプリケーションのコマンドの見通しが悪くなり、ユーザーから分かりにくくなります。また、ユーザーがメニュー項目を選択したときにダイアログ ボックスを表示するようにすれば、ダイアログ ボックスに対して追加の設定を行えるため、メニューの階層化を最小限に抑えることができます。メニュー バーに余裕がないときや、あまり使わないメニュー コマンドがあるとき、あるいは他のメニュー コマンドとの関係を強調したいときなどに、階層化したメニューを使ってください。

▼ サブメニューを作成するには

1. 上位のレベルのメニュー項目を作成します。
2. このメニュー項目の下に、サブメニューに含まれる (下位の階層の) メニュー項目を追加します。追加するメニュー コントロールは、<→> ボタンを使って、1 レベル分多くインデントします。

   1 レベル分インデントすると、ドットがさらに2 つ (..) 表示されます。インデントレベルを戻すには、<←> を使ってください。
3. さらに下位の階層にメニューを追加するときは、2 で作成したメニュー項目に対して、同じことを繰り返します。

8.10　アクセス キー，メニュー コントロール，タイトル バーの色設定

SCREEN オブジェクトの ControlPanel プロパティを使うと、メニュー，アクセス キー，タイトル バーなどの色を変更することができます。色の設定方法の詳細については、第 14 章「アプリケーションの外観の変更」を参照してください。

注　意　プログラミング環境およびフォーム デザイナの [補助] メニューの [画面設定] コマンドを使うと、デザイン時の画面の色を設定できます。これらの設定はプログラマがプログラミングするときのためのもので、アプリケーションの実行時の色設定には影響しません。アプリケーションの実行時の色を設定するには、必ず **SCREEN** オブジェクトの ControlPanel プロパティを使います。ControlPanel プロパティは、デザイン時には設定できず、必ずコード中から設定を行います。

8.11 トップ レベルのメニュー コントロールに対する Click イベント プロシージャ

一般に、タイトル バーに表示されるメニュー タイトルであるメニュー コントロールは、下位のメニューを表示するためだけの機能を持ちます。また、サブメニューを持つメニュー項目についても、同じことが言えます。これらのメニュー コントロールをユーザーがクリックしたとき、メニューをドロップダウンする動作は Visual Basic が自動的に行うため、プログラマがコードを記述する必要はありません。このように、クリックすると自動的に別のメニューがドロップ ダウンするメニュー コントロールを、トップレベルのメニュー コントロールと呼びます。

メニュー タイトルや、サブメニューを持つメニュー項目も 1 つのメニュー コントロールであり、これらに対して Click イベント プロシージャを記述してもかまいません。通常は、そのコントロールのクリックによってドロップダウンするメニューの、初期化処理を記述します。ただし、ここに記述できるコードには制限があります。トップレベルのメニュー コントロールに対するイベント プロシージャに記述できるコードは、その下位のメニュー コントロールのプロパティ (Caption , Checked , Visible , Enabled , Separator) を設定するものだけです。それ以外の操作を行った場合、結果は予測できず、アプリケーションの実行に重大な障害が生じる可能性があります。特にイベントを発生させるようなコードは、たとえ単純な Paint イベントであっても、確実に副作用を生じます。また、そのメニュー コントロールと同レベル以上の階層のメニュー コントロールに影響を与えるようなコードを書くこともできません。例えば、メニュー タイトルとして [ファイル] メニューと [編集] メニューがある場合、[ファイル] メニューに含まれるメニュー コマンドから、[編集] メニューを変更することはできません。ただし、上位のレベルのメニュー コントロールから、下位のメニュー コントロールを変更することは可能です。また、すべてのレベルのメニュー コントロールのプロパティを調べることは問題ありません。このような問題はサブメニューを持つメニュー コントロールにだけ生じ、サブメニューを持たない通常のメニュー項目では、記述するコードに特に制限はありません。

8.12 実行時のメニュー コントロールの追加と削除

メニューの内容は、デザイン時で設定できるだけでなく、必要に応じて実行時にもメニュー コントロールを追加や削除が行えます。次の例では、ユーザーがウィンドウをオープンすると、オープンしたファイルのリストが [ウィンドウ] メニューに追加されます。

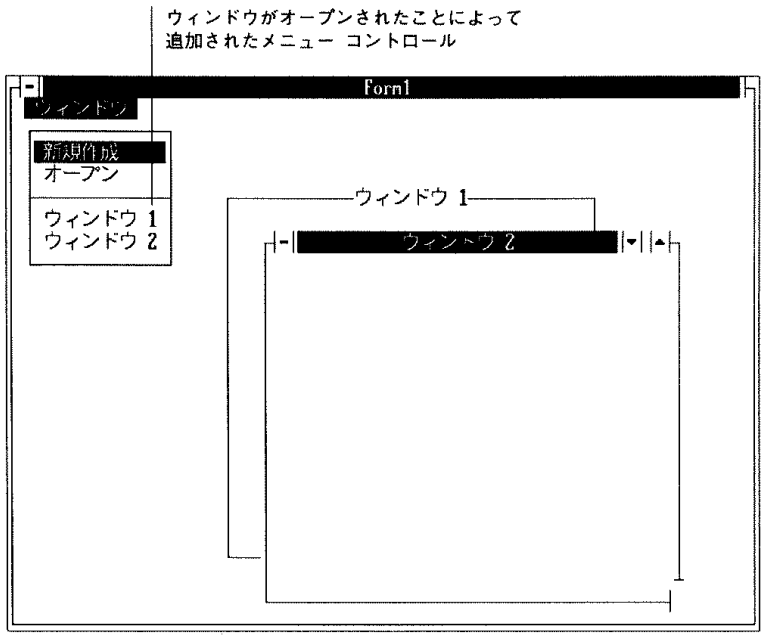


図 8.4　実行時のメニュー項目の追加

この例のメニューのセパレータ バーより下の部分には、そのときオープンされているウィンドウに応じて、必要なメニュー項目が表示されます。ウィンドウをオープンするとメニュー項目が増え、ウィンドウをクローズするとメニュー項目が減ります。実行時にメニュー コントロールを追加または削除するには、コントロール配列を使います。コントロール配列の詳細については、第 4 章「コントロールの作成と利用」を参照してください。

他のコントロール配列と同じように、メニュー コントロール配列は、共通の名前を持ち、イベント プロシージャを共有するメニュー コントロールの集まりです。コントロール配列のどの要素であるかは、インデックス番号 (Index プロパティの値) によって示されます。コントロール配列のうちの 1 つがイベントを認識すると、Visual Basic は引数としてインデックス番号をイベント プロシージャに引き渡します。イベント プロシージャでこのインデックス番号をチェックすれば、各要素に合わせて異なる処理を行うことができます。

同じコントロール配列に含めたいメニュー コントロールは、同じメニューに含まれている必要があります。メニューの階層の中で同じレベルにあっても、別のメニューであれば、1 つのコントロール配列に含めることはできません。

▼ メニュー コントロール配列を作成するには

1. フォーム デザイナでメニュー デザイン ウィンドウをオープンします。
2. メニュー コントロールを作成し、Caption プロパティや CtlName プロパティを設定します。
3. [インデックス番号] テキスト ボックスに対して、インデックス番号 (Index プロパティの値) を入力します。

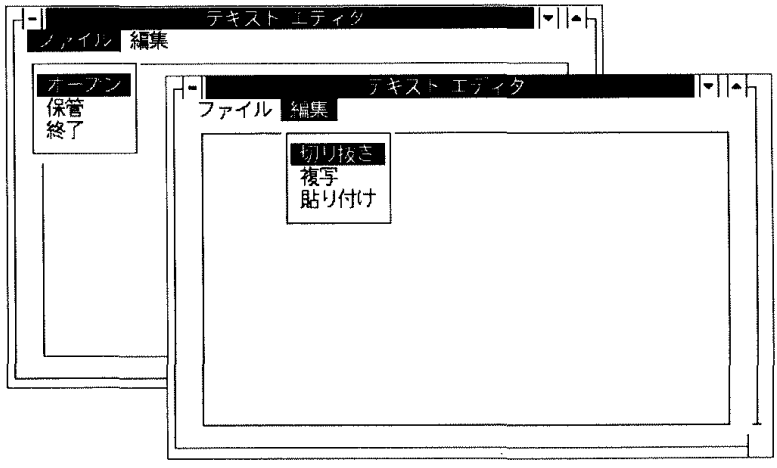
1 つのメニュー コントロールに対してインデックス番号を割り当てれば、その時点でコントロール配列が生成されます。

追加や削除する必要のないメニュー項目は、デザイン時にすべて設定してもかまいません。この場合、メニュー デザイン ウィンドウで、必要な設定を行います。

4. コード中から、**LOAD** ステートメントを使って、コントロール配列の要素を追加します。同じコントロール配列に含めたいメニュー コントロールは、同じコントロール名 (CtlName プロパティの値) でなければならないことに注意してください。インデックス番号は必ずしも連続している必要はありませんが、複数のコントロールに同じインデックス番号を割り当てることはできません。

実行時にメニュー コントロールを追加するテキスト エディタの例

次に示すシンプルなテキスト エディタは、実行時にコントロール配列を使ってメニュー コントロールを追加しています。アプリケーションのメインのフォームのメニュー バーには、[ファイル] と [編集] の 2 つのメニュー タイトルが表示されています。[ファイル] メニューには、[オープン], [保管], [終了] コマンドが含まれます。また [編集] メニューには、[切り抜き], [複写], [貼り付け] コマンドが含まれます。



ユーザーがファイルをオープンするたびに、新しい要素がメニュー コントロール配列に追加され、そのキャプションにファイル名が設定されます。ファイル名は [ファイル] メニューの下の方に表示されます。また、ユーザーが最初にファイルをオープンしたとき、他のメニュー項目とファイル名を分けて表示するためのセパレータ バーが追加されます。

コントロール配列の名前 (CtlName プロパティの値) は mnuFileArray です。デザイン時にフォーム デザイナでこのコントロールを作成し、インデックス値を 0 に設定します。これによってコントロール配列が作成され、実行時にコントロールを追加できるようになります (デザイン時に 0 以外のインデックス番号の要素を作成してもかまいませんが、通常はインデックス番号 0 の要素にします)。コントロール配列の要素の追加は、[ファイル] メニューの [オープン] コマンドに対するイベント プロシージャで行います。

実行時にメニュー コントロールを追加するには、次のように **LOAD** ステートメントを使い、引数にインデックス番号を指定します。

```
LOAD mnuFileArray(Index)
```

LOAD および **UNLOAD** ステートメントで指定するインデックス番号を管理するために、この例では変数 NewItem を定義して、要素を追加するたびにこの値に 1 を足しています。同じインデックス番号を別の要素に割り当てようとすると、エラーが発生します。変数 NewItem は、mnuFileOpen_Click イベント プロシージャで、静的 (STATIC) 変数として定義されます。

```
STATIC NewItem AS INTEGER
```

このアプリケーションでは、単純化するために、ファイル，ドライブ，ディレクトリの各リスト ボックスは使っていません。その代わり **INPUTBOX$** 関数を使い、オープンするファイルの指定をユーザーが直接入力するようになっています。またこのアプリケーションでは、特にエラー処理を行っていません。これらの機能については、第 10 章「ファイルシステム コントロールの利用」と、第 9 章「エラー ハンドリング」を参照してください。

▼ テキスト エディタ (EDITOR.FRM) を作成するには

1. プログラミング環境で [ﾌｧｲﾙ] メニューの [新規フォーム] コマンドを選び、ファイル名を “EDITOR.FRM” と指定して、フォーム デザイナを起動します。
2. フォーム デザイナで、テキスト ボックスを含むフォームを作成します。
3. 次の表に従って、フォームとテキスト ボックスのプロパティを設定します。

オブジェクト	プロパティの設定
フォーム	FormName = Form1 Caption = テキスト エディタ WindowState = 2 (最大化)
テキスト ボックス	CtlName = Text1 Text = (なし) Multiline = True

4. [ｳｨﾝﾄﾞｳ] メニューの [メニュー デザイン] コマンドを選択します (フォームが最大化されているときは、[ﾒﾆｭｰ] キーを押すとメニュー バーが表示されます)。
5. メニュー デザイン ウィンドウで、アプリケーションのメニューを作成します。プロパティは次のように設定してください。チェックは “X” マークが表示されている状態、未チェックは表示されていない状態を示します。セパレータと表示の欄には、デフォルトと違う設定をするものだけ示してあります。

キャプション (Caption)	コントロール名 (CtlName)	インデント	セパレータ (Separator)	表 示 (Visible)	インデックス番号 (Index)
&F/ファイル	mnuFile	しない			
&O/オープン	mnuFileOpen	レベル 1			
&S/保管	mnuFileSave	レベル 1			
&X/終了	mnuFileExit	レベル 1			

(なし)	mnuSep	レベル 1	チェック	未チェック
(なし)	mnuFileArray	レベル 1		未チェック 0
&E/編集	mnuEdit	しない		
&T/切り抜き	mnuCut	レベル 1		
&C/複写	mnuCopy	レベル 1		
&P/貼り付け	mnuPaste	レベル 1		

6. メニュー デザイン ウィンドウをクローズします。[ﾌｧｲﾙ] と [編集] の 2 つのメニュー タイトルがフォームの一番上に表示されます。

7. プログラミング環境で、フォーム モジュール (EDITOR.FRM) のモジュールレベルに次の行を追加します。

```
CONST True = -1
CONST False = 0
```

8. [編集] メニューの [イベント プロシージャ] コマンドで表示されるリスト ボックスで、[オブジェクト] リスト ボックスから mnuFileOpen、[イベント] リスト ボックスから Click をそれぞれ選択し、<編集> ボタンをクリックします。

9. mnuFileOpen_Click イベント プロシージャに次のコードを追加します。

```
SUB mnuFileOpen_Click ()
    STATIC NewItem AS INTEGER   'インデックス番号を管理する変数
    NL$ = CHR$(13) + CHR$(10)           '改行コード
    openfile$ = INPUTBOX$("オープンするファイル:")
                                        'ファイル名の取得
    OPEN openfile$ FOR INPUT AS #1      'ファイルのオープン
    DO UNTIL EOF(1)
        LINE INPUT #1, getline$         'ファイルの読み込み
        TextIn$ = TextIn$ + getline$ + NL$
                        '改行コードを追加しながら文字列変数に代入
    LOOP
    Text1.Text = TextIn$            'テキスト ボックスに表示
    CLOSE #1
    mnuSep.Visible = True               'セパレータ バーを表示
    'メニューに追加される最初の項目であれば、要素の追加は行わない
    IF NewItem = 0 THEN
        mnuFileArray(0).Caption = openfile$
        mnuFileArray(0).Visible = True
    ELSE
        LOAD mnuFileArray(NewItem)
                                'コントロール配列に要素を追加
        mnuFileArray(NewItem).Caption = openfile$
                                'キャプションをファイル名に設定
        mnuFileArray(NewItem).Visible = True
                                'ファイル名を表示
    END IF
    NewItem = NewItem + 1       'インデックス番号に 1 をプラス

END SUB
```

このプロシージャは、ユーザーに対して、オープンするファイル名の入力を要求し、その名前を変数 openfile$ に格納します。**OPEN** ステートメントでこのファイルを読み込み用にオープンし、**LINE INPUT#** ステートメントを使ってファイルの内容を 1 行ずつ読み込みます。1 行を読み込むと、行末にキャリッジ リターン/ライン フィードが付加されます。すべて読み終わったら、ファイルの内容をテキスト ボックスに表示します。

ファイルをオープンして読み込むと、コントロール配列の新しい要素のキャプション (Caption プロパティの値) は、そのファイル名になります。まず、セパレータバーの Visible プロパティが True (-1) に設定され、メニュー上に表示されるようになります。最初にオープンしたファイルは、デザイン時にフォーム デザイナで設定した、インデックス番号 0 の要素に割り当てられ、新たに配列に要素を追加する操作は行いません。2 つ目のファイルを読み込むときは、**LOAD** ステートメントを使ってコントロール配列に要素を追加しています。最後に、コントロール配列の新しい要素の Visible プロパティを True (-1) に設定します。以上の操作によって、メニューにセパレータ バーとファイル名を示すメニュー コントロールが追加されます。

10. 次のmnuFileArray_Click イベント プロシージャは一度ロードされたファイルをメニューから選択することによって再ロードします。

```
SUB mnuFileArray_Click(Index AS INTEGER)
    OPEN mnuFileArray(Index).Caption FOR BINARY AS #1
    length& = LOF(1)
    TextIn$ = SPACE$(length&)
    GET #1, , TextIn$
    Text1.Text = TextIn$
    CLOSE #1
END SUB
```

mnuFileOpen_Click イベント プロシージャでは **LINE INPUT#** ステートメントを使っていましたが、ここではバイナリ モードでオープンして **GET** ステートメントを使っています。**LINE INPUT#** ステートメントが文字列を行単位で読み込むのに対し、**GET** ステートメントは 1 度の操作でファイル全体を読み込んでいます。ここでは、**GET** ステートメントが変数の長さの分だけのバイト数を 1 度に読み込むことを利用して、ファイルの長さと同じサイズのスペースだけの文字列を用意し、それを文字列変数に代入することによって、ファイル全体を 1 度に読み込んでいます。[オープン] コマンドでファイルをロードした場合と、メニュー上のファイル名を選んでロードした場合のスピードを比べてみてください。

OPEN ステートメントと **LINE INPUT** ステートメントの詳細については、オンライン ヘルプまたは 『リファレンス』 を参照してください。

11. mnuSave_Click イベント プロシージャに次のコードを追加します。

```
SUB mnuSave_Click
    openfile$ = INPUTBOX$ ("セーブするファイル名:")
    OPEN openfile$ FOR OUTPUT AS #1
    PRINT #1,Text1.Text
    CLOSE #1
END SUB
```

このプロシージャも **INPUTBOX$** を使って、保管するファイル名の入力をユーザーに求めます。ここでは **OPEN** ステートメントによって、読み込み用ではなく書き込み用としてファイルをオープンします。テキスト ボックスの現在の内容 Text1.Text を、**PRINT #** ステートメントでファイルに書き込みます。

12. mnuExit_Click イベント プロシージャに **END** ステートメントを追加します。

```
SUB mnuExit_Click ()
    SCREEN.HIDE
    END
END SUB
```

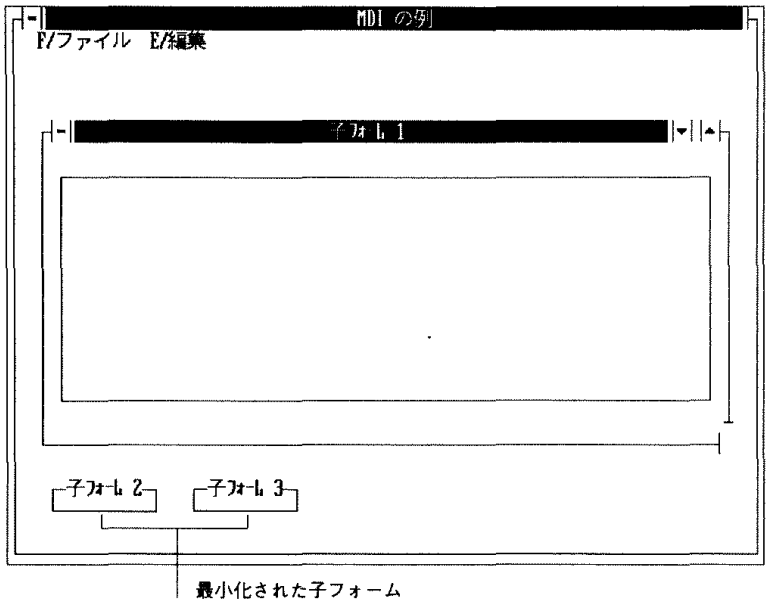
13. mnuEditCut_Click, mnuEditCopy_Click, mnuEditPaste_Click のそれぞれのイベント プロシージャにコードを追加します。

```
SUB mnuEditCut_Click ()
    CLIPBOARD.SETTEXT Text1.SelText
    Text1.SelText = ""
END SUB

SUB mnuEditCopy_Click ()
    CLIPBOARD.SETTEXT Text1.SelText
END SUB

SUB mnuEditPaste_Click ()
    temp$ = CLIPBOARD.GETTEXT()
    templen% = LEN(Text1.SelText)
    Text1.SelText = temp$
    Text1.SelStart = Text1.SelStart + Start + LEN(temp$)
      ⇒ - templen%
END SUB
```

テキスト ボックスは、最初からカット & ペーストの機能を持っており、[SHIFT]+[DEL], [CTRL]+[INS], [SHIFT]+[INS] などのキー操作に割り当てられています。ここでは、メニュー コマンドとして実現するために、あらためて **CLIPBOARD** オブジェクトを使ってプログラミングしています。**CLIPBOARD** オブジェクトを使って、テキストをカット，コピー，ペーストする方法については、第 7 章「テキストの処理」を参照してください。

8.13 マルチ ドキュメント インターフェイス(MDI)

Visual Basics では、1 つのフォームを親フォームとして、その中に別の複数のフォームを配置することができます。この場合、親に当たるフォームがアプリケーションのメインのウィンドウになり、子に当たるフォームのコンテナとして機能します。このように、1 つのウィンドウがコンテナとなり、複数のウィンドウで構成されるユーザー インターフェイスを、マルチ ドキュメント インターフェイス (MDI) と呼びます。また、親に当たるフォームを、コンテナ フォームまたは親フォームと呼びます。



Microsoft Word, Microsoft Works, Microsoft Visual Basic などのアプリケーションはマルチ ドキュメント インターフェイスを備えているので、1 つのアプリケーションで複数のドキュメント (ワード プロセッサ アプリケーションの文書, スプレッド シート アプリケーションのシートなど) を編集できます。

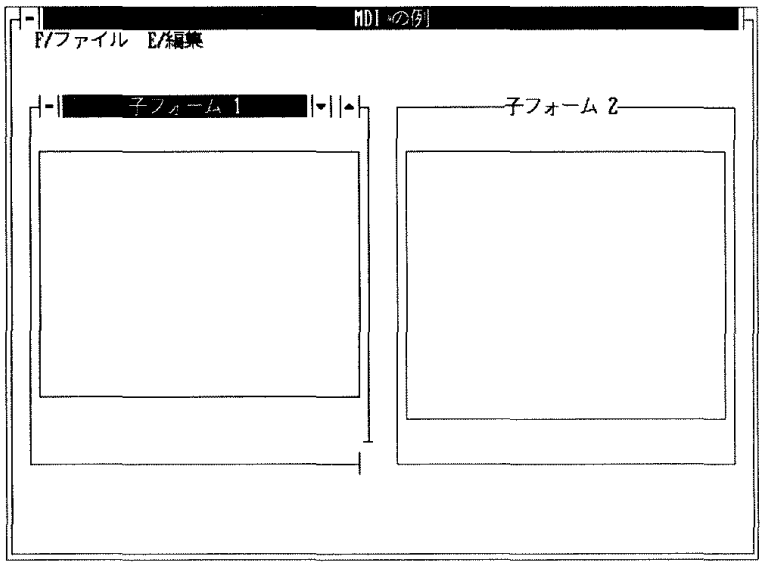
シングル ドキュメント インターフェイスのアプリケーションでは、そのフォームがアクティブであるときしか、フォーム上のコントロールとメニュー コマンドを利用できません。これは、別のフォームをアクティブにすると、新しいフォーム上のコントロールやメニュー コマンドがアクティブになるためです。この章の前の方で説明したテキスト エディタの例は、シングル ドキュメント インターフェイスです。[ﾌｧｲﾙ] メニューの [オープン] コマンドを選択すると、ファイルがオープンされ、同じフォーム上のテキスト ボックスにテキストが表示されます。

これに対し、MDI アプリケーションには、次のような特長があります。

- 1 つのアプリケーションは 1 つのコンテナ フォームを持ちます。他のフォームはすべて、コンテナ フォームの子フォームになります。
- コンテナ フォーム上に配置できるコントロールはメニュー コントロールだけです。子フォームには、コントロールを自由に配置できます。
- コンテナ フォームは常に最大化表示され、サイズを変更できません。
- コンテナ フォームには、タイトル バーとコントロール ボックスを含む境界線が表示されます。境界線は非表示にも設定できます。
- コンテナ フォームが、常にそのアプリケーションで最初に表示されるフォームになります。
- コンテナ フォームをアンロードすると、子フォームもすべて自動的にアンロードされます。逆に、子フォームをアンロードしてもコンテナ フォームはアンロードされません。

- 1つのフォーム モジュールに記録できるのは1つのフォームだけです。MDIアプリケーションのプロジェクトには、常に複数のフォーム モジュールが含まれることになります。プロジェクトの管理方法については、第6章「プロジェクトの管理」を参照してください。

メニュー バーはコンテナ フォームにだけ表示され、子フォームはメニュー バーを持ちません。コンテナ フォームのメニュー コマンドは、子フォームに対しても有効です。また、複数のフォームに別々にメニューが定義されている場合、そのときアクティブな子フォームのメニューが、コンテナ フォームに表示されます。



フォーム デザイナで、あるフォームのFormTypeプロパティを1 (MDI) に設定すると、そのフォームはMDIアプリケーションのコンテナ フォームとなります。アプリケーション中の他のフォームは、コンテナ フォームに対する子フォームになり、コンテナ フォームのクライアント領域 (タイトルとメニュー バーより下で、境界線の内側の部分) の中にだけ表示されます。

前のテキスト エディタの例をマルチ ドキュメント インターフェイス (MDI) に書き直すと、上の図のようになります。ユーザーの操作は、そのときアクティブな子フォームが対象になり、コンテナ フォームに表示されるメニューに関する操作も、アクティブな子フォームに対して有効になります。例えば、[ﾌｧｲﾙ] メニューの [保管] コマンドを実行すると、そのときアクティブな子ウィンドウで編集していたテキストが保管されます。

注 意 コマンドラインから、MDIアプリケーションの実行可能 (.EXE) ファイルを作成するときは、LINKのオブジェクト モジュール フィールドの先頭に、コンテナ フォームを含むフォーム モジュールを指定する必要があります。LINKユーティリティについては、オンライン ヘルプのLINKのトピックを参照してください。

MDI アプリケーションの作成

MDI アプリケーションを作成するには、デザイン時に MDI アプリケーションであることを指定しなければなりません。コンテナ フォームは、1 つのアプリケーションにつき 1 つだけです (設計時に指定します)。

メニューを使った MDI アプリケーションを作成する手順を次に示します。

1. フォームを作成し、フォーム デザイナでそのフォームの FormType プロパティを 1 (MDI) に設定します。このフォームが、コンテナ フォームになります。FormType プロパティを 1 に設定すると、そのフォームはすぐに最大化されます。
2. メニュー デザイン ウィンドウを使って、フォームにメニュー コントロールを追加します。この方法の詳細については、この章の前半を参照してください。
3. 子フォームを作成します。コンテナ フォームは 1 つだけなので、プロジェクトで作成する他のフォームはすべて自動的に子フォームとなります。
4. 子フォームにコントロールを追加します。子フォーム上に、コンテナ フォームとは別のメニュー コントロールを作成することもできます。この場合、子フォームがアクティブになると、子フォームに対して定義されているメニューがコンテナ フォームに表示されます。
5. フォーム デザイナのプロパティ バーを使って、コントロールのプロパティを設定します。
6. プログラミング環境に戻り、イベント プロシージャを記述します。このとき、コンテナ フォームを含むフォーム モジュールが、実行開始モジュールに設定されているかどうか必ず確認してください (必要に応じ、[実行] メニューの [実行開始ファイル設定] コマンドで、実行開始モジュールを変更してください)。アプリケーションの起動時に子フォームも表示したいときは、コンテナ フォームの Form_Load イベント プロシージャで、**SHOW** メソッドを使って表示します。

MDI テキスト エディタ アプリケーション

ここでは、さきほど作成したテキスト エディタ アプリケーション (EDITOR.FRM) を改造して、2 つあるウィンドウのどちらかを使ってテキストを編集する MDI アプリケーションを作成してみます。

このアプリケーションは、1 つのコンテナ フォームと 2 つの子フォームから構成されています。コンテナ フォームがアプリケーションの基本的な機能を提供し、2 つの子フォーム上で 2 つの異なるドキュメントを編集します。コンテナ フォームのメニュー バーに表示されるメニューは、そのときアクティブな子ウィンドウに対して有効です。

このアプリケーションでは、単純化するために、特にエラー処理を行っていません。エラー処理の方法については、第 9 章「エラー ハンドリング」を参照してください。

▼ テキスト エディタ アプリケーションを MDI に改造するには

1. さきほど作成したテキスト エディタ アプリケーションをロードします。プログラミング環境で、[ﾌｧｲﾙ] メニューの [プロジェクト オープン] コマンドを選択して、[ファイル名] テキスト ボックスに “EDITOR.FRM” と入力して、<了解> ボタンをクリックします。

2. [ﾌｧｲﾙ] メニューの [新規保管] コマンドを選択して、[ファイル名] テキスト ボックスに “PFORM.FRM” と入力して、<了解> ボタンをクリックします。このフォームが、MDI アプリケーションのコンテナ フォームになります。

3. [表示] メニューの [フォーム選択] コマンドを選ぶか、プロジェクト ウィンドウで <フォーム> ボタンをクリックし、フォーム デザイナに移ります。

4. テキスト ボックスにフォーカスを移してから [DEL] キーを押し、テキスト ボックスを削除します。

5. [F2] キーか [ﾒﾆｭｰ] キーを押してプロパティ バーを表示し、フォームの FormType プロパティを 1 (MDI) に設定します。また、フォームの他のプロパティを次のように設定します。

オブジェクト	プロパティ
コンテナ フォーム	FormName = PForm Caption = MDI テキスト エディタ FormType = 1 (MDI)

この段階では、EDITOR.FRM で作成したフォームと外見は変わりませんが、FormType プロパティを 1 (MDI) に設定したため、すでにコンテナ フォームとして扱われています。また、メニューの設定はそのまま残っています

6. 2 つの子フォームを作成します。フォームを作成するとき、1 つのフォームで 1 つのフォーム モジュールが構成されることに注意してください。このアプリケーションは 3 つのフォームを使いますので、フォーム モジュールも 3 ファイルになります。

[ﾌｧｲﾙ] メニューから [新規フォーム] コマンドを選択します。プロパティの設定を変更したため、保管するかどうかをたずねるダイアログ ボックスが表示されますので、<はい> を選んでください。新規に作成する子フォームには、Textwin1 と Textwin2 という名前をそれぞれ付けます。そのプロジェクトに含まれるフォームの一覧は、[表示] メニューの [フォーム選択] コマンドで確認できます。

フォームを作成したら、先ほどの図を参考に子フォームの位置とサイズを決めます。フォーム デザイナでは 1 度に 1 つのフォームだけが表示されるため、子フォームをデザインしているときにはコンテナ フォームは表示されませんが、コンテナ フォームは常に最大化されているため、画面全体をコンテナ フォームと見なすことができます。プロパティ バーの右端に表示される位置とサイズが参考になります。

子フォーム上には、それぞれ 1 つずつテキスト ボックスを配置します。また、子フォームとテキスト ボックスのプロパティは、次のように設定してください。両方のテキスト ボックスの設定はまったく同じであることに注意してください。

オブジェクト	プロパティ
左の子フォーム	FormName = Textwin1 Caption = 子フォーム 1
右の子フォーム	FormName = Textwin2 Caption = 子フォーム 2
左の子フォームのテキスト ボックス	CtlName = Text1 MultiLine = True Text = (なし)
右の子フォームのテキスト ボックス	CtlName = Text1 MultiLine = True Text = (なし)

7. [ﾌｧｲﾙ] メニューの [終了] コマンドを選び、フォーム デザイナを終了します。このとき、プロジェクト名の入力を求めるダイアログ ボックスが表示されますので、“MDIEDIT.MAK” と入力してください。

8. コンテナ フォームのモジュールレベルに次の行を追加します。[F2] キーを押すか [表示] メニューの [コード選択] コマンドを選び、PFORM.FRM が反転表示されている状態で [ENTER] キーを押します。

```
'コンテナ フォームから子フォームにアクセスできるようにします。
'$FORM TextWin1
'$FORM TextWin2
CONST True = -1
CONST False = 0
```

$FORM メタコマンドによって、コンテナ フォームのフォーム モジュール (PFOFRM.FRM) から、子フォームである Textwin1 および Textwin2 を操作できるようになります。フォーム モジュールのモジュール レベルには実行可能ステートメントを置くことはできませんが、このようにメタコマンドや宣言ステートメントは置くことができます。

9. Form_Load イベント プロシージャに次のコードを追加します。

```
SUB Form_Load()
    Textwin1.SHOW
    Textwin2.SHOW
END SUB
```

SHOW メソッドによって、2 つの子フォームが表示されます。

10. その他のイベント プロシージャのコードは、ほとんどそのまま利用できます。ただし、コード中でテキスト ボックスを指定するとき、どちらの子フォームのテキスト ボックスであるかを明確に指定する必要があります。そのときアクティブであるフォームを指定するには、**SCREEN** 特殊オブジェクトの ActiveForm プロパティを利用します。例えば、アクティブであるフォームに配置されているテキスト ボックスの、Text プロパティを参照するには、次のように指定します。

```
SCREEN.ActiveForm.Text1.Text
```

このアプリケーションでは、メニュー コントロール以外に使っているコントロールはテキスト ボックス (Text1) だけであり、コードが記述されているのは実行開始モジュールである PFORM.FRM だけであるため、PFORM.FRMを対象に Text1 を検索して SCREEN.ActiveForm.Text1 に置き換えると、簡単に書き換えることができます。

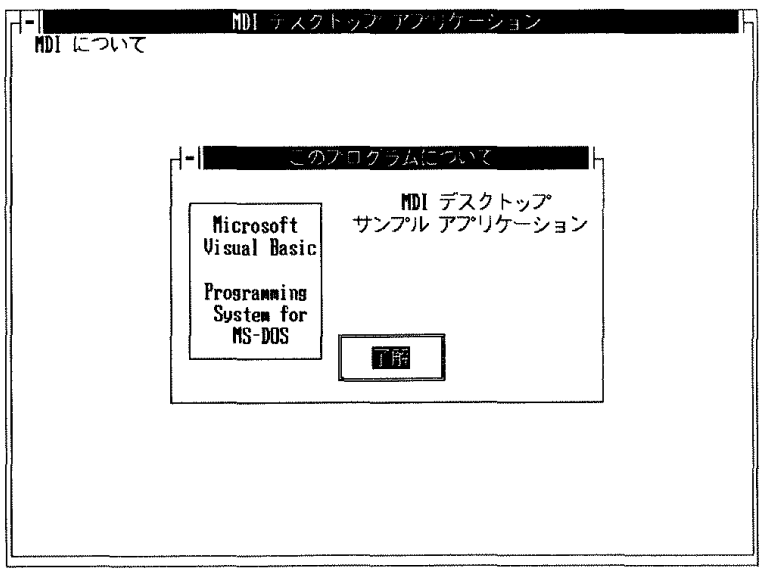
[検索] メニューの [置き換え] コマンドを選択し、[検索文字列] テキスト ボックスに "Text1"，[置換文字列] テキスト ボックスに "SCREEN.ActiveFrom.Text1" と入力して、置き換えを行ってください。<逐次確認> ボタンを選ぶと、確認しながら置き換えを行うことができます。

さらに、このアプリケーションでは、フォーカスを持っているオブジェクトが必ずテキスト ボックス (Text1) であることに注意してください。そのときフォーカスを持っているコントロールは、**SCREEN** 特殊オブジェクトの ActiveControl プロパティを使って参照できます。したがって、SCREEN.ActiveForm.Text1 を SCREEN.ActiveControl に置き換えても同じ意味になります。

MDI デスクトップ アプリケーションの作成

次のアプリケーションは、コンテナ フォームと子フォームとのさまざまな動作を示します。このアプリケーションは、コンテナ フォームと 、時計，電卓，メモ帳の 3 つのアプリケーションを表すアイコンから構成されます。

ユーザーがアイコンをダブルクリックすると、対応するアプリケーションのフォームがコンテナ フォーム上に復元されます。



時計，電卓，メモ帳アプリケーションは、Visual Basic のパッケージに含まれているサンプル アプリケーションを利用します。メモ帳アプリケーションは独自のメニューを持つので、メモ帳のアイコンをダブルクリックすると、コンテナ フォームのメニューがメモ帳のメニューで置き換えられます。

この 3 つの子フォーム以外にも、ユーザーが [MDI について] メニューを選択すると、コモン ダイアログボックス ツールキットを利用して、4 番目の子フォームの [MDI について] ダイアログ ボックスが表示されます。これはモーダル ダイアログ ボックスなので、他のフォームをアクティブにするには、<了解> ボタンをクリックしてクローズする必要があります (モーダル ダイアログ ボックスについては、第 5 章「ダイアログ ボックスの作成」を参照してください)。

ユーザーのマウス操作によって、アプリケーションは次のように動作します。

ユーザーがクリックする対象	アプリケーションの動作
時計アイコン，電卓アイコン	時計および電卓アプリケーションが復元されますが、コンテナ フォームのメニュー バーはそのままです。
メモ帳アイコン	メモ帳アプリケーションが復元され、コンテナ フォームのメニュー バーはメモ帳のメニュー バーに置き換えられます。
[MDI について] メニュー コマンド	[MDI について] ダイアログ ボックスが表示されます。他のフォームをアクティブにする前に、<了解> をクリックしてダイアログ ボックスをクローズする必要があります。

このアプリケーションでは、コモン ダイアログ ボックスのフォーム モジュール (CMNDLGF.FRM) とコード モジュール (CMNDLG.BAS) も使います。この 2 つのモジュールを使うと、あらかじめ用意されている汎用的なダイアログ ボックスを簡単に Visual Basic アプリケーションに追加できます。コモン ダイアログ ボックスに含まれているプロシージャを利用するには、[ファイル] メニューの [ファイル追加] コマンドを使って、コモン ダイアログ ボックスのフォームおよびコード モジュールをプロジェクトに追加してください。

▼ サンプルの MDI デスクトップ アプリケーションを作成するには

1. フォーム デザイナで コンテナ フォーム (MDI.FRM) を作成し、次の表に従ってプロパティを設定します。

オブジェクト	プロパティ
コンテナ フォーム	FormType = 1 (MDI) FormName = Mdi Caption = MDI デスクトップ アプリケーション

2. メニュー デザイン ウィンドウを使って、フォームに [MDI について] メニューを追加します。

コントロール	プロパティ
[MDI について] メニュー コントロール	Caption = &MDI について CtlName = mnuAbout

3. プログラミング環境で、[ﾌｧｲﾙ] メニューの [ファイル追加] コマンドを使って、次のモジュールをプロジェクトに追加します。

CALC.FRM	電卓アプリケーション
TOKEI.FRM	時計アプリケーション
CMNDLGF.FRM	コモン ダイアログ ボックスのフォーム モジュール
CMNDLG.BAS	コモン ダイアログ ボックスのコード モジュール
NOTEPAD.FRM	メモ帳アプリケーション

この他、必要なインクルード ファイル (.BI) がカレント ディレクトリにあることを確認してください。

4. 次のメタコマンドを、コンテナ フォームを含むフォーム モジュール (MDI.FRM) のモジュールレベルに追加します。

```
'$FORM frmCalc
'$FORM frmTokei
'$FORM frmCmnDlg
'$FORM frmSDINotePad
```

これらのメタコマンドによって、コンテナ フォームから子フォームにアクセスできるようになります。

5. mnuAbout_Click イベント プロシージャに次のコードを追加します。

```
SUB mnuAbout_Click ()
    msg$ = "MDI デスクトップ" + CHR$(13) + "サンプル アプリケーション"
    About msg$, 15, 3, 1
END SUB
```

このコードによって CMNDLG.BAS 中の About イベント プロシージャが呼び出され、[MDI について] ダイアログ ボックスに簡単なメッセージが表示されます。

6. MDI Form_Load プロシージャに次のコードを追加します。

```
SUB Form_Load ()
    frmCalc.SHOW
    frmCalc.WindowState = 1          'フォームを最小化します。
    frmTokei.SHOW
    frmTokei.WindowState = 1
    frmSDINotePad.Show
    frmSDINotePad.WindowState = 1
END SUB
```

このコードによって、アプリケーション起動時に時計，電卓，メモ帳アプリケーションを最小化します。ユーザーがアイコンをクリックしてアプリケーションを復元すると、WindowState プロパティは 0 (ノーマル) に設定されます。

メモ帳アプリケーション

この章の前半で作成した単純なテキスト エディタは、コントロール配列を使って、実行時にメニュー コントロールを追加する方法を示しています。このアプリケーションは Visual Basic で提供されているサンプルのメモ帳アプリケーションを単純化したものです。もとのアプリケーションのコードの一部を削除し、オープンするファイル名の入力を求めるときに、ファイル，ドライブ，ディレクトリ コントロールの代わりに **INPUTBOX$** 関数を使うなどの単純化が行われています。また、エラー処理の機能が削除されています。これらの機能を追加するには、第 10 章「ファイルシステム コントロールの利用」，第 9 章「エラー ハンドリング」，メモ帳サンプル アプリケーション (NOTEPAD.FRM) のコードを参照してください。

9 エラー ハンドリング

この章では、アプリケーションの実行中に発生したエラーの回避方法，識別方法，処理方法について説明します。このようなエラーは、“ランタイム エラー”と呼ばれます。ランタイム エラーは、プログラムの実行中に完了不可能な操作が行われた場合に起こり、通常は次のような状況で発生します。

- **ステートメントや関数の不正な実行**

  例えば、0 ～ 11535 の範囲の値だけが有効な **CHR$** 関数に対して負の値を指定しようとすると、エラーが発生します。

- **ファイル操作**

  例えば、特定の条件で、存在しないファイルをオープンしようとするとエラーが発生します。

- **ハードウェア的な制限**

  例えば、アプリケーション実行中にメモリが不足したり、ディスクの空き容量が不足するとエラーが発生します。このケースでは、アプリケーションからはエラーを回復できない場合があります。

9.1 エラー処理が必要な理由

ネットワーク デバイスの接続不良やフロッピー ディスクの入れ忘れなど、アプリケーション実行中にはさまざまな事態が発生する可能性があります。このような事態に対処するために、多くのアプリケーションではエラー処理の手順を用意する必要があります。

例えば次のプロシージャでは、拡張子が .TXT であるファイルがディスクに存在するかどうかをチェックしています。このプロシージャでは、**DIR$** 関数の戻り値が null 文字列 ("") かどうかはチェックしていますが、ランタイム エラーに対するエラー処理は考えられていません。

```
FUNCTION CheckTxtFiles%
    'DIR$ が指定したファイル名の条件と一致したときは
    '最初に一致したファイルの名前を返し、一致しないときは null 文字列 ("")
    'を返します。
    IF DIR$("*.TXT") <> "" THEN CheckTxtFiles% = NOT FALSE
        'ファイルが見つかったときは TRUE (-1) を返し、
        'そうでないときは FALSE (0) を返します。
END FUNCTION
```

このプロシージャは一見、**DIR$** 関数を呼び出したときに起こり得るどのような事態にも対応しているように見えます。しかし、カレント ドライブがフロッピー ディスク ドライブで、しかもフロッピー ディスクが正しくセットされていない場合は、“ディスクが準備されていません” エラーになり、アプリケーションは異常終了してしまいます。

アプリケーションをプログラミング環境から実行している場合は、その場でエラーの原因を取り除いて回復することも可能ですが、実行可能 (.EXE) ファイルを実行しているときは、エラー メッセージが表示されて実行が停止し、制御はコマンド ラインに戻ってしまいます。ここで再びアプリケーションを実行することはもちろん可能ですが、アプリケーションが異常終了したときに失われたデータがあれば、そのデータは回復できません。

このような場合は、適切なエラー処理ルーチンを記述して、アプリケーションが異常終了する前に正しい処置が取れるようにしておく必要があります。エラーが発生したときにエラー処理ルーチンに制御を移すことを、エラーを “トラップする” と言います。デバイスが原因で問題が生じた場合 (“ディスクが準備されていません” や “デバイスが準備されていません” エラーなど) に対応するエラー処理の例を次に示します。

プログラム例 9.1　ファイルのチェックにおけるエラー処理の例

```
FUNCTION CheckTxtFiles% (FileSpec AS STRING)
    'エラー発生したときに実行されるエラー処理ルーチンを設定します。
    ON LOCAL ERROR GOTO CheckError

    IF DIR$(FileSpec) <> "" THEN CheckTxtFiles% = NOT FALSE
    EXIT FUNCTION
        'EXIT FUNCTION によって、エラーが発生しなかった場合にエラー
        '処理ルーチンが実行されるのを防ぎます

    'エラーが発生したときは、ここに制御が移ります。
    'ERR 関数は、最後に発生したエラーのエラー コードを返します。
    'ERR の値:   71 = ディスクが準備されていません
    '            68 = デバイスが準備されていません
    CheckError:
    IF (ERR = 71) OR (ERR = 68) THEN
        MSGBOX "フロッピー ディスクを正しくセットしてください"
        RESUME
            'RESUME は、エラーが発生したステートメントに処理を戻します。
    ELSE
        'エラーを復旧できないときは、アプリケーションの実行を停止します。
        'ERR 関数はエラー コードを返します;
        'ERROR$ 関数はエラー メッセージを返します。
        MSGBOX "予想外のエラー (エラー コード : " + Str$(Err)
          ⇒ + " ) " + Error$ + "が発生しました。"
        END
    END IF
END FUNCTION
```

この例には、エラー処理で使われる典型的な関数とステートメントが含まれています。**ERR** 関数は、最後に発生したエラーを示すエラー コード (エラーの番号) を返します。**ERROR$** は、引数に指定したエラー コードが示すエラーが発生したときに、Visual Basic が表示するメッセージの文字列を返します。

RESUME ステートメントは、エラーの原因となったステートメントに処理を戻し、再度実行を試みます。ユーザーがエラーの原因を取り除いている場合はステートメントは問題なく実行されますが、そうでない場合は同じエラーが発生して、再びエラー処理ルーチンが呼び出されます。また、エラー処理ルーチンが対応していないランタイム エラーが発生したときは、**END** ステートメントの実行によってアプリケーションは終了します。

この例では、CheckError ラベルと **END FUNCTION** ステートメントの間のコードが“エラー処理ルーチン”です。このエラー処理ルーチンはプロシージャ内で定義されているため、“プロシージャ レベル”のエラー処理ルーチンです。

コード モジュールでは、エラー処理ルーチンをモジュール レベルで定義することも可能です。モジュール レベルでのエラー処理については、この章の「モジュール レベルのエラー処理」を参照してください。

9.2 エラー処理の方法

エラー処理には、次に示す 3 つのステップが必要です。ここでは主に、プロシージャ内で処理を行うプロシージャ レベルのエラー処理について説明します。

1. **ON LOCAL ERROR GOTO** *line* ステートメントを使って、ランタイム エラーが発生したときに実行するエラー処理ルーチンを設定し、エラー トラップを“有効”にします。

2. エラー処理ルーチンを記述します。エラーが発生してプログラムの制御がエラー処理ルーチンに移ると、そのエラー処理ルーチンは“アクティブ”になります。

   エラー処理ルーチンでは通常、**ERR** 関数の戻り値を基に **SELECT..CASE** ステートメントや **IF...THEN...ELSE** ステートメントでエラーの種類を判断し、それぞれに対して適切な処理を行います。前に説明した例では、“ディスクが準備されていません”または“デバイスが準備されていません”エラーが発生した場合は、ユーザーにフロッピー ディスクを正しくセットするように求めるメッセージを表示しています。それ以外のエラーが発生したときは、**ERR** 関数と **ERROR$** 関数を使い、エラーの種類を示すメッセージを表示します。Visual Basic のランタイム エラーの一覧は、『リファレンス』またはオンライン ヘルプを参照してください。

3. エラー処理ルーチンを終了します。

   前の例では、**RESUME** ステートメントによって、プログラムの制御をエラーが発生した位置に戻しています。この場合、エラーが発生したステートメントが再度実行されます。

エラー処理ルーチンの設定

エラー処理ルーチンを設定するには、**ON LOCAL ERROR GOTO** *line* ステートメントを使います。*line* には、エラー処理ルーチンの先頭の行を示す行番号かラベルを指定します。このステートメントを実行するとエラー トラップが有効になり、ランタイム エラーが発生するとエラー処理ルーチンに制御が移るようになります。エラー トラップ

は、**EXIT SUB, END SUB, EXIT FUNCTION, END FUNCTION** のいずれかのステートメントによって、そのプロシージャの実行が終了するまで有効です。

エラー トラップを再び無効にしたいときは、**ON LOCAL ERROR GOTO** *line* ステートメントで *line* (行番号) に 0 を指定します。実行するとすぐにエラー トラップは無効になります。エラー処理を無効にする方法について詳しくは、この章の「エラー処理を明示的に無効にする」を参照してください。

エラー処理ルーチンの記述

エラー処理ルーチンの先頭の行には、**ON LOCAL ERROR GOTO** *line* ステートメントの *line* で指定した行番号またはラベルを付けます。前のプログラム例では、エラー処理ルーチンの先頭を CheckError: というラベルで示しています。

エラー処理ルーチンは、通常のプログラムの流れでは実行されない位置に記述してください。エラー処理ルーチンは一般に、**EXIT SUB** ステートメントと **END SUB** ステートメントの間、または **EXIT FUNCTION** と **END FUNCTION** の間に置きます。エラー処理ルーチンの前に **EXIT SUB** または **EXIT FUNCTION** ステートメントがないと、エラー処理ルーチンは通常のプログラムの流れに含まれるため、エラーが発生していなくても実行されてしまいます。

RESUME を使ったエラー処理ルーチンの終了

エラー処理ルーチンの最後には、必ず **RESUME** ステートメントを置きます。**RESUME** ステートメントは、指定した引数によって、プログラム中のどこに制御を戻すかを示します。

ステートメント	機能
RESUME	エラーの発生したステートメントに戻り、実行を続けます。
RESUME NEXT	エラーの発生したステートメントの次のステートメントに戻り、実行を続けます。
RESUME *line*	*line*で指定した行番号またはラベルの付いたステートメントに戻り、実行を続けます。行番号は 0 以外でなければなりません。

RESUME および RESUME NEXT の利用

引数のない **RESUME** ステートメントは、エラーの発生したステートメントにプログラムの制御を戻して、そのエラーの原因となった操作を再度実行します。**RESUME NEXT** は、エラーの発生したステートメントの次のステートメントに制御を戻します。**RESUME** と **RESUME NEXT** の違いは次のようになっています。

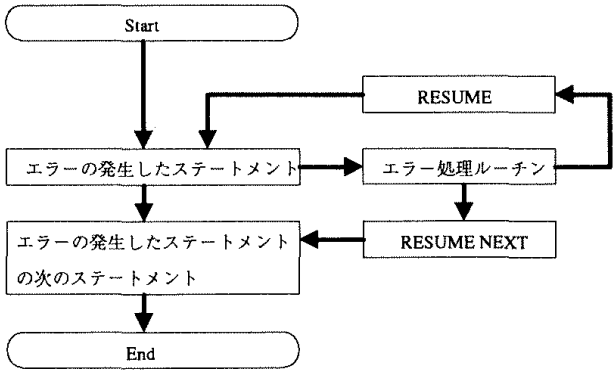


図 9.1 RESUME と RESUME NEXT のプログラム制御の流れ

RESUME と **RESUME NEXT** のどちらを使うかは、ユーザーが特に操作しなくてもエラーが回復できるかどうかによります。言い換えれば、エラー処理ルーチンでエラーの原因を解決するか、エラーの原因となったステートメントを飛ばして実行を続けるかのどちらかを選ぶことになります。ユーザーが (例えばフロッピー ディスクを正しくセットするなどの) エラーを回避する操作を行える場合は、通常は引数なしの **RESUME** を使います。

場合によっては、エラーを回復せずに無視して先に進んだ方がよいこともあります。例えば次のプログラム例では 2 つの変数の割り算を行っていますが、分子が 0 でなく分母が 0 のときは、“0 で除算しました” エラーが発生します。一方、分子と分母がともに 0 で、浮動小数点の割り算を行うと、“オーバーフロー” エラーが発生します。

この場合は、そのデータ型の最大数を返すことによって、無限大の代わりにするという対策をとることが可能です。次の例では、**RESUME NEXT** を使ってエラーの原因となったステートメントを回避しています。

プログラム例 9.2 RESUME NEXT を使ったエラー処理

```
FUNCTION DivideIt! (X AS SINGLE, Y AS SINGLE)
    ON LOCAL ERROR GOTO MathHandler
    DivideIt! = X/Y
    EXIT FUNCTION

    MathHandler:
    IF ERR = 11 OR ERR = 6 THEN
        DivideIt! = 3.4E38
            '“0 で除算しました” または “オーバーフロー” が発生した場合は
            '代わりに結果として無限大を返します。
    END IF

    'RESUME NEXT によって、EXIT FUNCTION から実行が継続されます。
    RESUME NEXT
END FUNCTION
```

RESUME *line* の利用

RESUME ステートメントで *line* 引数を指定すると、プログラムの制御を戻す位置を指定できます。例えば、ランタイム エラーの原因となったステートメントの前に戻って実行を継続したいときなどに、このステートメントを使います。

RESUME *line* ステートメントを使ったプログラムの例を次に示します。このプログラム例では、*line* 引数には StartHere ラベルを指定しています。ユーザーは VerifyFile プロシージャを使ってファイルの指定を確認することができます。**DIR$** は最初に一致したファイルのファイル名を返しますが、指定された文字列に一致するファイルが見つからないときは、null 文字列 ("") を返します。

注　意　次の例の VerifyFile プロシージャは、モジュール レベルで定義する必要のある複数の記号定数を使っています。Visual Basic では定数をローカルに定義することも可能ですが、通常はモジュール全体で利用できるようにモジュール レベルで定義します。また、インクルード ファイル中で定義し、**$INCLUDE** メタコマンドを使ってインクルードすると、すべてのモジュールで同じ定数を共有できます。

プログラム例 9.3　RESUME line を使ったエラー処理

```
DECLARE FUNCTION VerifyFile () AS STRING
'各定数は、モジュール レベルで定義するか、インクルード ファイルで定義して
'インクルードする必要があります。
    CONST Err_BadFileName = 52, Err_DriveDoorOpen = 71
    CONST Err_DeviceUnavailable = 68, Err_InvalidFilename = 64
    CONST FALSE = 0, TRUE = NOT FALSE

FUNCTION VerifyFile () AS STRING
    NL$ = CHR$(13) + CHR$(10)        '改行とラインフィードの組み合わせ
    Prompt$ = "ファイルを指定してください。"

StartHere:
    Msg$ = Msg$ + Prompt$
    FileSpec$ = INPUTBOX$(Msg$)
    IF FileSpec$ = "" THEN EXIT FUNCTION
                                    '入力がないときは終了します

    ON LOCAL ERROR GOTO ErrorHandler
    VerifyFile = DIR$(FileSpec$)
    EXIT FUNCTION

ErrorHandler:
    SELECT CASE ERR
        'エラー コードの値に応じたメッセージを設定します
    CASE Err_InvalidFilename, Err_BadFileName
        Msg$ = "指定したファイル名は無効です。"
    CASE Err_DriveDoorOpen
        Msg$ = "フロッピー ディスクを正しくセットしてください。"
    CASE Err_DeviceUnavailable
        Msg$ = "指定したデバイスが見つかりません。"
    CASE ELSE
        Msg$ = ERROR$
    END SELECT
```

```
    Msg$ = Msg$ + NL$
    RESUME StartHere
                        '指定したラベルに制御を返して、ユーザーが
                        '別のファイル名を指定できるようにします
END FUNCTION
```

ユーザーが指定した(ワイルド カードを含んだりデバイスの指定だったりする)文字列に一致するファイル名が見つかると、最初に一致したファイル名がこの関数の戻り値になります。**DIR$** 関数でエラーにならない限り(一致するファイルがないために null 文字列を返す場合も含めて)、関数は正常に終了します。一方、ランタイム エラーが発生すると、Msg$ にメッセージを設定して、StartHere ラベルの行に制御が移ります。この結果、ユーザーは再度、ファイル名を示す文字列を入力できます。

ここで、エラーが発生する可能性があるステートメントの直前で、エラー トラップを有効にしている点に注意してください。エラーが発生しなかった場合、このトラップは **EXIT FUNCTION** が実行されると同時に解除されます。エラーが発生する可能性があるステートメントが限られている場合は、その場所でだけエラー処理を有効にするようにします。

呼び出しパス

“呼び出しパス”は、現在実行中の位置に到達するまでの実行経路を示す、それまでに呼び出されたプロシージャのリストです。最後に呼び出されたプロシージャがリストの先頭になります。呼び出しパスを調べるには、コール ヒストリ ウィンドウが便利です。コール ヒストリ ウィンドウの 1 行目のプロシージャが、呼び出しパスの最初のプロシージャです。

エラー処理における呼び出しパスの利用

それぞれのプロシージャで、発生する可能性のあるすべてのランタイム エラーに対する処理を記述すると、同じコードが複数存在することになり、効率的ではありません。そこで、プロシージャ内部のエラー処理ルーチンでは、そのプロシージャだけで発生するエラーの処理を行い、より一般的なエラー処理については、呼び出し元のプロシージャで処理することができます。

次の例は、この呼び出しパスを使ったエラー処理を行う、CheckFile プロシージャのエラー処理ルーチンを示しています。このエラー処理ルーチンでは、1 つのエラーに対する処理だけが含まれています。ほかのエラーに対しては、エラーを疑似的に発生させる **ERROR** ステートメントを実行し、別の場所で設定されているエラー処理ルーチンを検索させるようにしています。**ERR** 関数は直前に発生したエラーのエラー コードを示すため、**ERROR ERR** を実行すると、そのエラー処理ルーチンに制御が移ったきっかけとなったエラーを再び疑似的に起こすことになります。エラー処理ルーチン内でエラーが発生しても、そのエラー処理ルーチンが再び呼び出されることはなく、呼び出しパスをさかのぼって、有効になっているがアクティブではないエラー処理ルーチンが順に検索されるようになっています。次の例では、CheckFile プロシージャを呼び出す前に AddData プロシージャがエラー処理ルーチンを有効に設定しているため、CheckFile プロシージャ内のエラー処理ルーチンでエラーが発生すると、プログラムの制御は AddData のエラー処理ルーチンに移ります。

プログラム例 9.4　呼び出しパスを利用したエラー処理

```
SUB AddData (FileName AS STRING)
    ON LOCAL ERROR GOTO ErrorHandler1
Start:
    FileName = INPUTBOX$("ファイル名を入力してください。")
    TrueFlag% = CheckFile(FileName)
    ...
    EXIT SUB

ErrorHandler1:
    SELECT CASE ERR
        CASE 68
            Msg$ = "ドライブが無効です。"
        CASE 52, 64
            Msg$ = "ファイルの指定が無効です。"
        CASE ELSE
            Msg$ = STR$(ERR) + ": " + ERROR$
        END SELECT

    MSGBOX Msg$
    RESUME Start
END SUB

FUNCTION CheckFile% (FileName AS STRING)
    'ファイルが存在するときは -1 (TRUE) を、
    'そうでないときは 0 (FALSE) を、それぞれ返します。
    ON LOCAL ERROR GOTO ErrorHandler2
    CheckFile% = 0
    IF DIR$(FileName) <> "" THEN CheckFile% = -1
    EXIT FUNCTION

ErrorHandler2:
    IF ERR = 71 THEN
        MSGBOX "フロッピー ディスクを正しくセットしてください。"
    ELSE
        ERROR ERR
        'エラーを疑似的に発生させて、呼び出し元のエラー処理ルーチンに
        '処理を移します。
    END IF

    RESUME
END FUNCTION
```

9.3　モジュール レベルのエラー処理

コード モジュールでは、モジュール全体に対して有効な、モジュール レベルのエラー処理ルーチンを設定することができます。これによって、複数のプロセス間で同じエラー処理ルーチンを共有することが可能です。

ただしフォーム モジュールでは、モジュール レベルに実行可能ステートメントを記述できないため、モジュール レベルでエラー処理ルーチンは設定できません。

モジュール レベルのエラー処理ルーチンの設定

モジュール レベルのエラー処理ルーチンを設定するには、エラー処理ルーチンをモジュール レベルに記述します。エラー処理ルーチンの終了条件は、アプリケーションの実行を終了するか、制御をプロシージャに戻すか、エラーを再度発生させるかのいずれかになります。アプリケーションの実行を終了するには、**END** ステートメントを実行します。プログラムの制御をプロシージャに戻すには、プロシージャ レベルのエラー処理ルーチンと同様に **RESUME** または **RESUME NEXT** ステートメントを実行します。エラーを再度発生させるには、**ERROR ERR** ステートメントを実行します。

例えば、次のようなステートメントをモジュール レベルに記述します。

```
ModuleErrorHandler:
    MSGBOX ERROR$
    RESUME NEXT
```

このエラー処理ルーチンに対して、モジュール内の必要なプロシージャから、**ON ERROR GOTO** *line* ステートメントを使ってトラップを設定します。プロシージャレベルのエラー処理ルーチンを設定するときと異なり、**LOCAL** キーワードが指定されていないことに注意してください。**ON LOCAL ERROR GOTO** *line* ステートメントは、プロシージャ レベルの (ローカルな) エラー処理ルーチンを設定するのに対し、**ON ERROR GOTO** *line* ステートメントはモジュール レベルのエラー処理ルーチンの設定を行います。

```
FUNCTION CheckFile% (FileName AS STRING)
    ON ERROR GOTO ModuleErrorHandler
    CheckFile% = 0
    IF DIR$(FileName) <> "" THEN CheckFile% = -1
END FUNCTION
```

コード モジュールのモジュール レベルにエラー処理ルーチンを記述する場合、そのルーチンが通常の処理の流れの中に入らないように注意してください。一般的には、コードの最後 (**END** ステートメントの後) に記述します。

プロシージャ レベルとモジュール レベルのエラー処理の使い分け

1 つのアプリケーションの中で、モジュール レベルのエラー処理ルーチンとプロシージャ レベルのエラー処理ルーチンを使い分けることができます。例えば、あらかじめ予測可能なエラーについてはプロシージャ レベルのルーチンで処理し、一般的なエラーについてはモジュール レベルのルーチンで処理するようにします。

次の例では、ProcLevel1 プロシージャでモジュール レベルのエラー処理を有効に設定してから、ProcLevel2 を呼び出しています。ProcLevel2 プロシージャで有効になっているエラー処理ルーチンには、1 種類のエラーに対する処理しか記述されていません。ProcLevel2 の **DIR$** 関数でエラーが発生すると、プロシージャ内のエラー処理ルーチン (LocalErrorHandller) に処理が移りますが、**ERR** 関数の戻り値が 71 でなければ、**ERROR ERR** ステートメントが実行されます。このようにすると、呼び出しパスをさかのぼって有効でありアクティブでないエラー処理ルーチンの検索が行われ、ProcLevel2 プロシージャを呼び出した ProcLevel1 が調べられます。ここでエラー処理ルーチンが有効に設定されているため、プログラムの制御はモジュールレベルのエラー

処理ルーチンの先頭行である ModuleErrorHandler: の行に移ります。その後、**RESUME NEXT** ステートメントによって、プログラムの制御は ProcLevel2 の中のプロシージャレベルのエラー処理ルーチンに戻ります。

プログラム例 9.5　プロシージャ レベルとモジュール レベルのエラー処理の使い分け

```
'モジュール レベルの最後に置く。
ModuleErrorHandler:
    MSGBOX ERROR$
    RESUME NEXT

SUB ProcLevel1 ()
    ON ERROR GOTO ModuleErrorHandler
    ProcLevel2
END SUB

SUB ProcLevel2 ()
    FileName$ = INPUTBOX$("ファイル名を入力してください。")
    ON LOCAL ERROR GOTO LocalErrorHandler
    FileSpec$ = DIR$(FileName$)
    EXIT SUB

LocalErrorHandler:
    IF ERR = 71 THEN
        MSGBOX "フロッピー ディスクを正しくセットしてください。"
        RESUME
    ELSE
        ERROR ERR
    END IF
END SUB
```

注　意　エラー処理ルーチンの設定や戻り位置の指定には行番号や行ラベルが使われますが、別のモジュールの行番号や行ラベルを指定することはできません。

呼び出しパスからのエラー処理ルーチンの検索方法

ランタイム エラーが発生すると、呼び出しパスに従って、制御を移すエラー処理ルーチンが検索されます。次のような原則があります。

- 呼び出しパスの検索では、イベントの発生によって起こった呼び出しは無視されます。これは Visual Basic のイベントだけでなく、従来の Basic で使っていた割り込みによるイベントに関しても同様です。この制限があるため、イベント プロシージャ内のエラー処理ルーチンは、呼び出しパスを検索することによって他のモジュールのエラー処理ルーチンへ処理を移すことができません。
- エラー処理ルーチンの中でエラーが発生しても (意図的に発生させたのかどうかにかかわらず)、そのエラー処理ルーチンに再帰的に制御が移ることはありません。つまり、アクティブになっているエラー処理ルーチンには制御は移りません。エラーが発生すると、呼び出しパスをさかのぼって、有効でありアクティブでないエラー処理ルーチンが検索されます。

呼び出しパスからの検索は、次のような順番で行われます。

1. プロシージャ中でエラーが発生したとき、そのプロシージャで有効になっているアクティブでないエラー処理ルーチンがあれば、それが実行されます。
2. なければ、呼び出しパスをさかのぼって検索が行われます。エラーが発生したプロシージャを呼び出したプロシージャでエラーが発生していると見なして、1 から繰り返されます。このとき、エラーが発生したプロシージャが別のモジュールから呼び出されている場合は、別のモジュールを検索する前に、エラーが発生したプロシージャを含むモジュールのモジュール レベルが先に検索されます。また、呼び出しパスに含まれるプロシージャをすべて検索しても見つからないときにも、モジュール レベルが検索されます。
3. 呼び出しパスに含まれるすべてのプロシージャとモジュール レベルを検索して、有効でありアクティブでないエラー処理ルーチンが見つからない場合は、エラーメッセージを表示してプログラムは終了します。

エラー処理ルーチンからの復帰

有効なエラー処理ルーチンが見つかると、プログラムの制御はそのルーチンに移り、実行が継続されます。エラー処理ルーチンの位置と、エラー処理の最後に実行される **RESUME** ステートメントの指定によって、プログラムの実行がどこから再開されるかが決まります。

表 9.1 RESUME および RESUME NEXT の後のプログラムの処理の流れ

ステートメント	レベル	動作
RESUME	プロシージャ レベル	エラー処理ルーチンを含むプロシージャで最後に実行されたステートメントに戻って実行を続けます。
	モジュール レベル	エラー処理ルーチンを含むモジュールで最後に実行されたステートメントに戻って実行を続けます。
RESUME NEXT	プロシージャ レベル	エラー処理ルーチンを含むプロシージャで最後に実行されたステートメントの次のステートメントに戻って実行を続けます。
	モジュール レベル	エラー処理ルーチンを含むモジュールで最後に実行されたステートメントの次のステートメントに戻って実行を続けます。

プロシージャ レベルのエラー トラップは、次のいずれかが実行されると無効になります。

ON LOCAL ERROR GOTO 0, END SUB, END FUNCTION, EXIT SUB, EXIT FUNCTION

呼び出しパスに基づいた処理の続行

アプリケーションの呼び出しパスが複雑で、プロシージャ レベルのエラー処理ルーチンがいくつも存在する場合は、エラーが発生したときにどのエラー処理ルーチンが呼ばれ、その実行後にどこに戻るかを十分に把握しておく必要があります。予想外の位置のエラー処理ルーチンが実行されたり、処理後に制御が戻ったりすると、何が問題かが分かりづらいエラーの原因になります。

次の例では、ProcLevel3 プロシージャで発生したランタイム エラーによって、処理の流れが ProcLevel1 のエラー処理ルーチンに戻っています。エラー処理ルーチンで **RESUME NEXT** ステートメントが実行されると、処理の流れは ProcLevel1 にとどまります。この結果、ProcLevel2 の **MSGBOX** ステートメントは実行されません。つまり、呼び出しパスに従ってエラー処理ルーチンを検索した結果、処理の流れは ProcLevel3 から ProcLevel1 に直接移っています。

プログラム例 9.6　呼び出しパスに基づいた処理の続行

```
SUB ProcLevel1 ()
    ON LOCAL ERROR GOTO LocalErrorHandler
    ProcLevel2
    EXIT SUB

LocalErrorHandler:
    RESUME NEXT
END SUB

SUB ProcLevel2 ()
    ProcLevel3
    MSGBOX "ProcLevel2 の最後"
END SUB

SUB ProcLevel3 ()
    ERROR 1
END SUB
```

複雑な構成のプログラムのためのガイドライン

複数のフォームおよびコード モジュールを含む大きなアプリケーションの場合は、エラー処理も非常に複雑になる可能性があります。大規模なアプリケーションを作成するときは、次に示すガイドラインに従うことをおすすめします。

- すべてのエラー処理ルーチンに **ERROR ERR** ステートメントを入れます。これにより、発生したエラーに対応する処理がエラー処理ルーチンに用意されていない場合でも、他のエラー処理ルーチンの検索が行われます。また、この処置を行うことによって、Visual Basic がエラー メッセージを表示するのは、アプリケーション全体を通して処理できないエラーが発生したときに限られることになります。このテクニックは、アプリケーションをテストするときに、処理できないエラーを明らかにするのにも役立ちます。

- **ERROR ERR** ステートメントを使わないときは、すべてのエラー処理ルーチンが最終的に呼び出すことができるような一般的なエラー処理用のプロシージャを用意します。例えば、アプリケーションを終了する前にユーザー データを保存する処理などを記述します。
- エラーの処理を行いたくない場合は、**END** ステートメントで強制的に処理を終了します。

9.4 エラーの集中処理

それぞれのプロシージャの中で、すべてのエラーに対するエラー処理を行おうとすると、同じエラー処理用のコードを繰り返し記述しなければならない場合があります。このような場合は、適切なエラー処理用のプロシージャを作成することによって、プログラムのサイズとプログラミングの手間を減らすことができます。

次の例の FileErrors プロシージャは、このようなエラー処理用のプロシージャの例です。ここでは、発生したエラーの種類を表すメッセージを表示すると同時に、エラーに対応して行う操作を選ぶためのボタンを表示します。ユーザーが選んだボタンは値に変換されて、プロシージャの呼び出し元に返されます。

注 意 この例で使っている Err_DeviceUnavailable などの定数は、モジュール レベルと必要なすべてのプロシージャ レベルで定義されていなければなりません。

プログラム例 9.7 汎用性のあるエラー処理ルーチン

```
FUNCTION FileErrors% (errVal AS INTEGER)
'パラメータ: errVal は、ERR 関数が返す値です

'戻り値        意味
'==================================
'0             RESUME
'1             RESUME NEXT
'2             エラーを処理できません
'3             予期しないエラー

SELECT CASE errVal
    CASE Err_DiskMedia                    'エラー番号 72
        Msg$ = "フロッピー ディスクが損傷している可能性があります。"
        msgType% = 5                      '<再試行>, <取消> ボタン
    CASE Err_DeviceUnavailable            'エラー番号 68
        Msg$ = "デバイスが準備されていない可能性があります。"
        msgType% = 5
    CASE Err_DiskNotReady                 'エラー番号 71
        Msg$ = "フロッピー ディスクを正しくセットしてください。"
        msgType% = 5
    CASE Err_DeviceIO                     'エラー番号 57
        Msg$ = "内部的なディスク エラーです。"
        msgType% = 5
    CASE Err_DiskFull                     'エラー番号 61
        Msg$ = "ディスクの空き容量が不足です。続行しますか?"
        msgType% = 5
```

```
    CASE Err_BadFileName, ErrBadFileNameOrNumber
                                        'エラー番号 64, 52
        Msg$ = "ファイル名または番号が不正です。"
        msgType% = 1                    '<了解>, <取消> ボタン
    CASE Err_PathDoesNotExist   , Err_FileNotFound
                                        'エラー番号 75, 53
        Msg$ = "パスまたはファイルが見つかりません。"
        msgType% = 5
    CASE Err_BadFileMode                'エラー番号 54
        Msg$ = "指定されたアクセス モードでオープンできません。"
        msgType% = 5
    CASE Err_FileAlreadyOpen            'エラー番号 55
        Msg$ = "ファイルはすでにオープンされています。"
        msgType% = 5
    CASE Err_InputPastEndOfFile         'エラー番号 62
        Msg$ = "ファイルの最後より後を読み込もうとしました。"
        msgType% = 5
    CASE ELSE
        FileErrors% = 3
        EXIT FUNCTION
END SELECT

response% = MSGBOX (Msg$, msgType%, "ディスク エラー")

SELECT CASE response%
    CASE 1, 4               '<了解> ボタン, <再試行> ボタン
        FileErrors% = 0
    CASE 5                  '<無視> ボタン
        FileErrors% = 1
    CASE 2, 3               '<取消> ボタン, <中止> ボタン
        FileErrors% = 2
    CASE ELSE
        FileErrors% = 3
END SELECT
END FUNCTION
```

この FileErrors プロシージャは、ファイルとディスクに関する一般的なエラーを処理します。ディスク I/O に関連しないエラーが発生した場合は、プロシージャは値として 3 を返します。この場合は、FileErrors を呼び出したプロシージャは自分自身でエラーを処理するか、別のプロシージャを呼び出す必要があります。

注　意　アプリケーションが大規模になるにつれて、同じ定数を複数のモジュールで使う必要が出てきます。このような場合、その定数の宣言をインクルード ファイルで行うようにすると、各モジュールで同じ宣言を繰り返し入力する必要がなくなります。

ディスクへの読み書きを行うすべてのプロシージャで FileErrors プロシージャを利用すれば、アプリケーションのエラー処理を単純化できます。また、FileErrors のような汎用的なプロシージャを独立したコード モジュールにしておけば、別のアプリケーションでも利用可能です。さらに、エラーの処理を集中化することによって、同じエラーに対して常に同じメッセージ ボックスが表示されるようになり、エラーの処理を一定した方法で行えるようになります。

次の例の ConfirmFile プロシージャでは、前の例の汎用的なエラー処理ルーチン FileErrors を使ってエラーを処理しています。このプログラム例では、FileErrors の戻り値に応じて ConfirmFile が次に行う動作が決定されます。

プログラム例 9.8 汎用的なエラー処理ルーチンの呼び出し

```
FUNCTION ConfirmFile% (TheName AS STRING, Operation AS INTEGER)
    ON LOCAL ERROR GOTO ConfirmFileError
                                        'エラーのトラップを有効にする。
    TheFile$ = DIR$(TheName)
                                        'ファイルが存在するかどうか調べる。
    ON LOCAL ERROR GOTO 0               'エラーのトラップを無効にする。
    ...
    EXIT FUNCTION

ConfirmFileError:
    Action% = FileErrors%(ERR)
    SELECT CASE Action%
        CASE 0
            RESUME
        CASE 1
            RESUME NEXT
        CASE 2
            EXIT FUNCTION
        CASE ELSE
            ERROR ERR
    END SELECT
END FUNCTION
```

FileErrors と ConfirmFile の 2 つのプロシージャは、第 10 章「ファイルシステム コントロールの利用」と第 12 章「ファイルとデバイスの操作」で参照するプログラム例の中でも使われています。

9.5 エラー処理を後で実行

アプリケーションによっては、エラーをトラップしてもいったんプログラムの実行を続け、後で処理したい場合があります。Visual Basic では、エラー状況の記録だけを行ってプログラムを中断しないようにする場合、**ON [LOCAL] ERROR RESUME NEXT** ステートメントを使います。

次の例は、このようにエラー処理を後で実行する方法を示しています。エラーが発生しても、**ON LOCAL ERROR RESUME NEXT** ステートメントによって長い計算をそのまま続け、エラーのチェックは、この計算が終了した後で行っています。エラーが発生したかどうかは、**ERR** 関数が返す値を調べれば分かります。**ERR** 関数の値が 0 でなければエラーが発生したことを示しており、その値に従って適切なエラー処理ルーチンを実行することができます。

プログラム例 9.9　エラー処理を後で実行

```
'エラーが発生しても、アプリケーションの実行を中断させません。
ON LOCAL ERROR RESUME NEXT
Condition% = False
DO UNTIL Condition% = TRUE
    '長い計算をループの中で実行し、Condition% が TRUE になったら
    'ループを終了します。
    ...
LOOP

'ここではじめてエラーが発生したかどうかをチェックし、必要に応じて
'エラー処理を実行します。
SELECT CASE ERR
    CASE 0
        'エラーは発生していません。
    CASE 6                  'エラー番号 6 は“オーバーフロー”です。
        '“オーバーフロー”エラーの処理をここに記述します。
    CASE 11                 'エラー番号 11 は“0 で除算しました”です。
        '“0 で除算しました”エラーの処理をここに記述します。
    CASE ELSE
        'その他のエラーの処理をここに記述します。
END SELECT
'エラー処理を行った後の処理を、この後に記述します。
...
```

以前に示したプログラム例 9.3 を **ON LOCAL ERROR RESUME NEXT** ステートメントを使って書き直したものを次に示します。

```
...
ON LOCAL ERROR RESUME NEXT
DO
    ERR = 0      'ERR 関数の戻り値を初期化します。
    VerifyFile = DIR$(FileSpec$)

    IF ERR THEN
        SELECT CASE ERR
                    'エラー コードを調べてメッセージを作成します。
        CASE Err_InvalidFilename, Err_BadFilename
            Msg$ = "指定したファイル名は無効です。"
        CASE Err_DriveDoorOpen
            Msg$ = "フロッピー ディスクを正しくセットしてください。"
        CASE Err_DeviceUnavailable
            Msg$ = "指定したデバイスが見つかりません。"
        CASE ELSE
            TmpErr% = ERR
            ON LOCAL ERROR GOTO 0
                    'ON LOCAL ERROR RESUME NEXT を OFF にします。
            ERROR TmpErr%
                    'エラー処理ルーチンの内部でエラーを発生させます。
        END SELECT
    END IF
    MSGBOX Msg$
LOOP UNTIL ERR = 0
END FUNCTION
```

CASE ELSE ブロックの中で、いったん **ON LOCAL ERROR GOTO 0** ステートメントを使ってローカル エラー処理を無効にしてからエラーを再度発生させている点に注意してください。このステートメントによって **ON ERROR RESUME NEXT** を無効にしておかなければ、Visual Basic は **ERROR** ステートメントによって発生させた疑似的なエラーを無視してしまいます。

このほか、発生が予想されるエラーを無視する目的で、**ON LOCAL ERROR RESUME NEXT** ステートメントを使うこともできます。次に示すプログラム例では、**NAME** ステートメントを使ってファイル名を変更し、バックアップ ファイルを作成していますが、操作の対象となるファイルが存在しない可能性もあります。このため、ファイルが存在する場合はファイル名を変更しますが、存在しない場合は発生するエラーを無視します。

```
...
ON LOCAL ERROR RESUME NEXT
        'NAME ステートメントで発生する可能性のあるエラーを無視します。

NAME "ThisFile.Dat" AS "ThisFile.Bak"
ERR = 0             '予想したエラーの処理のロジックを混乱させないために、
                    'ERR 関数の戻り値を初期化します。
OPEN "ThisFile.Dat" FOR RANDOM AS #1
IF ERR THEN
    SELECT CASE ERR
...
```

このプログラム例では、**ON LOCAL ERROR RESUME NEXT** ステートメントによって、エラーが発生しても処理を中断しないようにしておいて、IF ERR THEN 以降の部分で、**OPEN** ステートメントのエラーをチェックしています。**NAME** ステートメントで発生したエラーを無視するために、**NAME** ステートメントのすぐ後で**ERR** 関数の戻り値をリセットしています。

エラー処理を後で実行するためのガイドライン

エラー処理を後で実行するときは、次に示す 4 つの点に注意してください。

- エラー処理を後で実行する場合は、これまで説明してきたエラー処理ルーチンと異なり、ラベルや行番号でエラー処理ルーチンの先頭行を指定しません。また、エラー処理ルーチンの最後に **RESUME** ステートメントがありません。**ERR** 関数の戻り値を基に、条件判断を行っているだけです。
- **ERR** 関数が返すエラー コードは、最後に発生したエラーのものです。もし複数のエラーが発生していても、以前のエラー コードを得ることはできません。
- **RESUME, RESUME NEXT, ON ERROR** のいずれかのステートメントが実行されると、Visual Basic は **ERR** 関数の戻り値を 0 にリセットします。それまでの **ERR** 関数の戻り値を後で参照したい場合は、これらのステートメントのいずれかを実行する前に、値を別の変数に保存しておく必要があります。
- エラーの発生と、そのエラー処理の間に、別の処理を行う場合は特に注意が必要です。エラー処理を行う前に別のエラーが発生すると、**ERR** 関数の戻り値が変わってしまうため、エラー処理が正しく行われません。

モジュール レベルでエラー処理を後で実行

ON ERROR RESUME NEXT ステートメントは、モジュール レベルで実行することもできます。機能は、次に示すモジュール レベルのエラー処理ルーチンと、それを有効にする **ON ERROR GOTO** *line* を実行することと同じです。

```
ModuleLevelHandler:
    RESUME NEXT

ON ERROR GOTO ModuleLevelHandler
```

エラーが発生して、このエラー処理ルーチンに制御が移っても、エラーが発生したステートメントの次のステートメントが実行されるだけで、それ以外の処理は行われません。このテクニックはプログラムのサイズが小さいときは問題ありませんが、サイズが大きくなるにつれて実際的ではなくなってきます。これは、あらゆる種類のエラーが発生する可能性のあるステートメントをすべて予測し、そのステートメントの後で **ERR** 関数の戻り値を調べて対応する処理ルーチンを実行する必要があるためです。

9.6　エラー処理を明示的に無効にする

ON [LOCAL] ERROR GOTO *line* で有効に設定したエラー処理を無効にするには、**ON [LOCAL] ERROR GOTO 0** ステートメントを使います。このステートメントは、**ON [LOCAL] ERROR RESUME NEXT** ステートメントで有効に設定したエラー処理も、すべて無効にします。

プロシージャ レベルのエラー処理を無効にする

プロシージャ レベルのエラー処理ルーチンを無効にするには、**ON LOCAL ERROR GOTO 0** ステートメントを使います。このステートメントは、そのとき有効になっている、そのプロシージャで設定されたプロシージャ レベルのエラー処理ルーチンを無効にします。別のプロシージャで設定されたエラー処理ルーチンには影響しません。

エラー処理ルーチンの中で **ON LOCAL ERROR GOTO 0** ステートメントを実行すると、**ERROR ERR** を実行したときと同じく、エラーが再度発生します。この場合、アクティブになっているエラー処理ルーチンを再帰的に呼び出すことはできないため、呼び出しパスから検索が行われます。

モジュール レベルのエラー処理を無効にする

モジュール レベルのエラー処理ルーチンを無効にするには、**ON ERROR GOTO 0** ステートメントを使います。このステートメントは、そのとき有効になっているモジュール レベルのエラー処理ルーチンを無効にします。このステートメントは、モジュール内のどのプロシージャ中でも実行できます。

エラー処理ルーチンの中で **ON ERROR GOTO 0** ステートメントを実行すると、**ERROR ERR** を実行したときと同じく、エラーが再度発生します。この場合、アクティブになっているエラー処理ルーチンを再帰的に呼び出すことはできないため、呼び出しパスからの検索が行われます。

9.7 エラー処理の別の利用法

場合によっては、エラーの原因となるユーザーからの入力を、エラーとしてでなく有効なデータとして処理する方が便利なこともあります。このような場合は、プログラムの制御にエラー処理を利用すると、すべてのデータを同じ方法で処理できるようになります。また、アプリケーションに必要な情報を取得する手段としてエラー処理を利用することもできます。

エラー処理をプログラムの制御に利用する

ON LOCAL ERROR RESUME NEXT ステートメントを使ってプログラムの流れを制御するテクニックを、プログラム例 9.10 に示します。0 または π の整数倍の値のアーク タンジェントを計算して "0 で除算しました" エラーが発生すると、エラー処理ルーチンは単に次のステートメントを実行します。ここで **ERR** 関数の戻り値を調べて、0 以外の値を返しているときは、数学的に不定の操作が行われていると判断して、**ERR** 関数の戻り値に応じて適切なエラー処理を実行します。

プログラム例 9.10 エラー処理をプログラムの制御に利用する

```
SUB ArcTanTest ()
    Number% = VAL(INPUTBOX$("数値を入力してください:"))
    ON LOCAL ERROR RESUME NEXT
    ArcTangent = 1 / (TAN(Number%))
    Msg$ = STR$(Number%) + " のアークタンジェントは "

    IF ERR THEN
        '得られた値が不定 (無限大) であることをユーザーに知らせます。
        Msg$ = Msg$ + "未定義"
    ELSE
        '得られた値が実数のときは、その値を表示します。
        Msg$ = Msg$ + STR$(ArcTangent)
    END IF

    MSGBOX Msg$
END SUB
```

エラー処理を情報の取得に利用する

エラー処理は、他の方法では得られない情報を取得するための手段として使うこともできます。例えば、そのとき使っているハードウェアがサポートしているスクリーン モードを調べる場合に、このテクニックを使うことができます。**SCREEN** ステートメントを実行したときに、引数で指定したスクリーン モードがサポートされていなければエラーが発生します。このエラーを利用して、ハードウェアがサポートしているすべてのスクリーン モードを調べることができます。

ユーザー定義のエラー コード

ERROR ステートメントを使って、Visual Basic for MS-DOS で使われていないエラー コードに対して、アプリケーション独自のランタイム エラー コードを割り当てることができます。Basic では 1 ～ 32767 の範囲の値がエラー コードとして利用できますが、実際に使われているのは、比較的小さい番号だけです。

注　意　Basic であらかじめ定義されているエラー コードとの干渉を避けるため、エラー コードをユーザーが定義するときは、32767 から降順に番号を割り当ててください。

9.8　エラー処理を含むコードのコンパイル

RESUME または **RESUME NEXT** ステートメントを含むプログラムを、コマンド ラインから BC.EXE を使って直接コンパイルするときは、次に示すエラー処理オプションのいずれかを指定する必要があります。

- 次のステートメントを使っているときは、必ず /E オプションを指定します。

  ON [LOCAL] ERROR GOTO

  RESUME *line*

- 次のステートメントを使っているときは、必ず /X オプションを指定します。

  RESUME [0]

  RESUME NEXT

  ON [LOCAL] ERROR RESUME NEXT

注　意　これらのステートメントを使っているプログラムを、/E または /X の各オプションを指定せずにコンパイルしようとすると、コンパイル エラーが発生します。

---

10 ファイルシステム コントロールの利用

多くのアプリケーションでは、ディスク上のファイルを指定するときに、ドライブ名，ディレクトリ名，ファイル名の一覧を画面に表示し、この中からファイルを選ぶようになっています。Visual Basic では、このようなファイルシステムのユーザー インターフェイスを簡単に作成できるように、ドライブ リスト ボックス，ディレクトリ リスト ボックス，ファイル リスト ボックスの 3 つの専用のコントロールを用意しています。この章では、これらのファイルシステム コントロールを使って、アプリケーション内でディスク ファイルの表示や選択を行う方法を説明します。

10.1 ファイルシステムのためのコントロール

フォーム デザイナでファイルシステム コントロールを配置するには、ツールボックスまたは [ツール] メニューの、[ディレクトリ リスト], [ドライブ リスト], [ファイル リスト] を選択します。次のように表示されます。

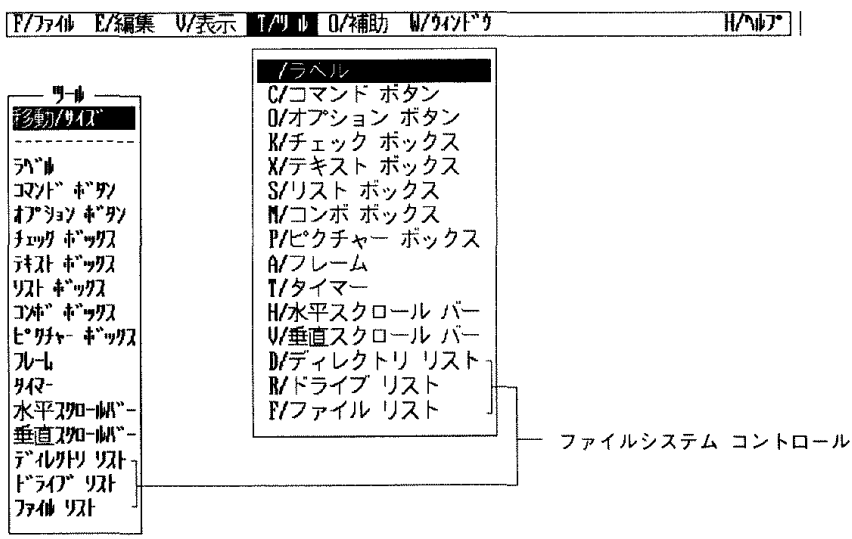


図 10.1 ファイルシステム コントロールの選択

これらのコントロールは、柔軟性が高い標準的なファイルシステムのユーザー インターフェイスを、簡単なプログラミングだけで実現できるように設計されています。各コントロールは、ファイルに関する情報を自動的に取得するので、基本的な動作に対するプログラミングは必要ありません。外観のカスタマイズや表示する情報の指定は、他のコントロールと同様にプロパティで設定します。これらのコントロールは、単独

でも使えますし、組み合わせて使うこともできます。組み合わせて使う場合は、それぞれのコントロールが互いに連携して動作するように、イベント プロシージャにコードを記述します。

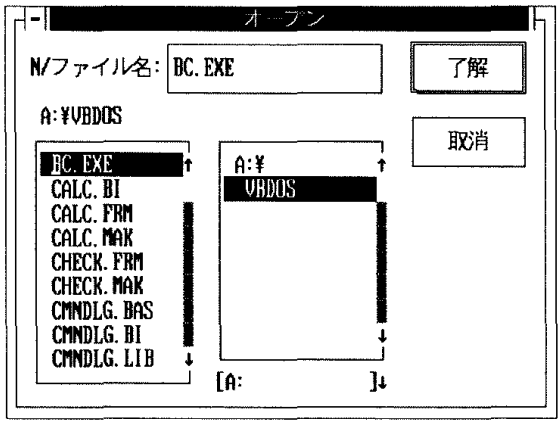


図 10.2　ファイルシステム コントロールを使ったダイアログ ボックス

これらのコントロールは、必要なすべての情報をオペレーティング システムから取得しています。コントロールが取得した情報にコード中からアクセスするには、各コントロールのプロパティを使います。表示する内容も、プロパティを使って指定できます。例えばファイル リスト ボックスには、デフォルトで、カレント ディレクトリ (アプリケーションを起動したディレクトリ、または **CHDIR** ステートメントで設定されたディレクトリ) の内容が表示されます。ファイル リスト ボックスには、ワイルドカードを使って、特定のパターン (例えば *.FRM) に一致する名前を持つファイルの一覧を表示することもできます。ワイルドカードを使ったファイル名の指定は、ファイル リスト ボックスの Pattern プロパティを使います。例えば次のように指定すると、拡張子 .FRM を持つファイルだけをファイル リスト ボックスに表示することができます。

```
File1.Pattern = "*.FRM"
```

ドライブ リスト ボックス

ドライブ リスト ボックスは、ドロップダウン リスト ボックスの一種で、そのとき選択されているドライブを示します。カレントのドライブを変更するには、コントロールがフォーカスを持っているときに、ドライブ名を示す文字を入力します。また、ドライブ リスト ボックスの右側に表示されている矢印をクリックすると、そのとき有効なすべてのドライブの一覧を含むリスト ボックスがドロップ ダウンしますから、その中から選択することもできます。選択すると、そのドライブ名がドライブ リスト ボックスのテキスト ボックス部分に表示されます。

そのとき選択されているドライブを知るには、ドライブ リスト ボックスの Drive プロパティを調べます。また、次のように指定すると、コード中からドライブを変更できます。

```
Drive1.Drive = "A:"
```

ドライブ リスト ボックスでドライブを変更しても、MS-DOS レベルのカレント ドライブは変更されません。MS-DOS レベルでカレント ドライブを変更するには、

CHDRIVE ステートメントを使います。次のように Drive1.Drive を引数として **CHDRIVE** ステートメントを実行すると、Visual Basic で作成したアプリケーション上でのドライブの指定が、MS-DOS レベルでのカレント ドライブになります。

```
CHDRIVE Drive1.Drive
```

ドライブ リスト ボックスの Drive プロパティは、フォーム デザイナ上で設定することはできません。

ディレクトリ リスト ボックス

ディレクトリ リスト ボックスには、そのとき選択されているドライブのディレクトリ構造が、あるディレクトリを中心に表示されます。表示の中心となるディレクトリを、"展開されているディレクトリ" と呼びます。Path プロパティは、そのとき展開されているディレクトリの名前を示しています。デフォルトでは、カレント ディレクトリが展開されているディレクトリです。ディレクトリ リスト ボックスには、このディレクトリを中心に、その直下にあるサブディレクトリの一覧と、展開されているディレクトリからルート ディレクトリまでのディレクトリの一覧が表示されます。インデントによって、ディレクトリ構造の深さが分かるようになっています。

ディレクトリ リスト ボックスでは、表示されている項目 (ディレクトリ名) のうちの 1 つが、常に反転表示されています。ディレクトリ リスト ボックスがフォーカスを持っているとき、ユーザーが [↑] または [↓] キーを押すと、反転表示されている項目が上下に移動します。このとき、どの項目が反転表示されているかを示すのが、ListIndex プロパティです。ListIndex プロパティが -1 なら、そのとき展開されているディレクトリ自身が反転表示されています。その直下のディレクトリには、ListIndex 0 から順番に 1, 2, 3 と値が割り当てられます。また、そのとき展開されているディレクトリのすぐ上のディレクトリは ListIndex -2、2 つ上のディレクトリは ListIndex-3 というように、ルート ディレクトリに到達するまで順番に番号が割り当てられます。

Path プロパティに別のディレクトリのパスを設定すると、そのディレクトリを中心に展開された状態に表示が変わります。例えば上の図で、Path プロパティに "C:¥PROJECTS" という文字列を代入すると、PROJECTS サブディレクトリを中心に展開された状態が表示されます。このとき、ListIndex プロパティの値も割り当て直されます。Path プロパティの値が変更された直後では、新しく展開されたディレクトリが反転表示されますから、この場合では、ListIndex プロパティの値は -1 (PROJECTS サブディレクトリ) になります。この状態で C ドライブのルート ディレクトリをマウスでクリックして反転表示させたとすると、ListIndex プロパティの値は -2 になります。

次のようにすると、そのとき展開されているディレクトリが、ルート ディレクトリから見てどのくらい深いレベルにあるかを知ることができます。

```
GoHigher% = -1        ;そのとき展開されているディレクトリから始める.
;ディレクトリが存在しなければ、Dir1.List(x) は null 文字列を返す.
DO UNTIL Dir1.List(GoHigher%) = ""
    GoHigher% = GoHigher% - 1
LOOP
LevelsAbove% = ABS(GoHigher%)        ;正の数に変換する.
```

ドライブ リスト ボックスで別のドライブが選択されたとき、それと連動してディレクトリ リスト ボックスの表示が変わるようにするには、ドライブ リスト ボックスの Change イベントに対するイベント プロシージャに、次のように記述します。

```
Dir1.Path = Drive1.Drive
```

ドライブ リスト ボックスの Drive プロパティは、そのとき選択されているドライブ名を示しています。ディレクトリ リスト ボックスの Path プロパティにドライブ名だけを代入すると、そのドライブのデフォルトのディレクトリを中心に展開されます。デフォルトのディレクトリとは、MS-DOS 上で、例えば "D:" のように入力してカレント ドライブを変更したときにカレント ディレクトリとなるディレクトリです。ただし、Path プロパティの設定を変更して、ディレクトリ リスト ボックスに展開されるディレクトリを変更しても、MS-DOS レベルのカレント ディレクトリが変更されるわけではありませんから注意してください。MS-DOS レベルのカレント ディレクトリを変更するには、**CHDIR** ステートメントを使います。例えば、次のステートメントを実行すると、ディレクトリ リスト ボックスに展開されているディレクトリとカレント ディレクトリが一致します。

```
CHDIR Dir1.Path
```

Path プロパティは、デザイン時には設定できません。

ディレクトリ リスト ボックスに表示されている項目 (ディレクトリ名) の 1 つをユーザーがクリックすると、その項目が反転表示されます。つまり ListIndex プロパティの値が変わります。ダブル クリックすると、その項目 (ディレクトリ) を中心に表示されます。つまり、Path プロパティの値が変わります。

Listcount プロパティは、そのとき展開されているディレクトリの直下にあるサブ ディレクトリの個数を示しています。ディレクトリ リスト ボックスに表示されている項目の総数ではないことに注意してください。

ファイル リスト ボックス

ファイル リスト ボックスには、そのとき設定されているディレクトリに含まれるファイルの一覧が表示されます。デフォルトでは、このディレクトリはカレント ディレクトリです。表示するディレクトリは、ファイル リスト ボックスの Path プロパティで設定します。Path プロパティは、ディレクトリ リスト ボックスと同様に、デザイン時には設定できません。

ディレクトリ リスト ボックスで別のディレクトリが選択されたとき、それと連動してファイル リスト ボックスの表示が変わるようにするには、ディレクトリ リスト ボックスの Change イベントに対するイベント プロシージャに、次のように記述します。

```
File1.Path = Dir1.Path
```

特定の拡張子やベース名を持つファイルだけを表示したいときは、Pattern プロパティを使います。このとき、ワイルド カード文字 * と ? が使えます。例えば、"*.FRM" を Pattern プロパティに代入すると、拡張子 .FRM のファイルだけが表示されます。複数のパターンを組み合わせて指定したいときは、次のようにセミコロンで区切ります。

```
File1.Pattern = "*.FRM; *.BAS"
```

このステートメントでは、.FRM または .BAS のどちらかの拡張子を持つファイルが表示されます。

また、それぞれのファイルが持つ属性によって、表示するかしないかを指定することもできます。属性には、Archive, Normal, System, Hidden, ReadOnly があり、それぞれ同じ名前のプロパティが用意されています。例えば読み込み専用の属性を持つファイルだけを表示したいときは、ReadOnly プロパティだけを True (-1) に設定し、他の属性を示すプロパティを False (0) に設定します。

```
File1.ReadOnly = -1
File1.Archive = 0
File1.Normal = 0
File1.System = 0
File1.Hidden = 0
```

System プロパティのみ True に設定し、他のプロパティを False に設定すると、System 属性を持つファイルだけが表示されます。この場合、System と Hidden の両方の属性を持つファイルは、表示されません。そのファイルを表示したいときは、System と Hidden の両方のプロパティを True に設定する必要があります。反対に Hidden と System の両方のプロパティが True に設定された場合は、これらの属性のうちのいずれか一方だけを持つファイルは表示されません。すべてのファイルを表示するには、Normal, System, Hidden の 3 つのプロパティを True に設定する必要があります (Archive と ReadOnly の 2 つの属性は無視されます)。デフォルトでは、Archive, Normal, ReadOnly が True (-1)、System, Hidden が False (0) に設定されています。

複数のコントロールの同期

通常、ファイルシステム コントロールは、2 つ以上を組み合わせて同期させるようにします。例として、ドライブ リスト ボックス，ディレクトリ リスト ボックス，ファイル リスト ボックスがそれぞれデフォルト名である Drive1, Dir1, File1 と名付られているとすると、次のようにイベントが順に発生して行きます。

1. ユーザーが Drive1 ドライブ リスト ボックスからドライブを選択すると、Drive1 に対する Change イベントが発生します。
2. Drive1_Change イベント プロシージャに、次のようなコードを記述しておくようにします。

```
SUB Drive1_Change()
    Dir1.Path = Drive1.Drive
END SUB
```

新しく選択されたドライブの名前 (Drive1.Drive プロパティの値) が、Dir1 ディレクトリ リスト ボックスの Path プロパティに代入されると、そのドライブのデフォルトのディレクトリを中心に、Dir1 の表示が自動的に変更されます。このとき、Dir1 に対する Change イベントが発生します。

3. Dir1_Change イベント プロシージャに、次のようなコードを記述しておくようにします。

```
SUB Dir1_Change()
    File1.Path = Dir1.Path
END SUB
```

新しく選択されたディレクトリの名前 (Dir1.Path プロパティの値) が、File1 ファイル リスト ボックスの Path プロパティに代入されると、新しく設定されたディレクトリに含まれるファイルの一覧が File1 に自動的に表示されます。

10.2　ファイル検索アプリケーション

多くのアプリケーションでは、そのアプリケーションで必要なファイルを探して表示する機能を用意しています。ここでは、パッケージに含まれているSEEK サンプル アプリケーション (SEEK.FRM) を使って、ファイルシステム コントロールの利用方法を説明します。SEEK アプリケーションは、ユーザーが指定したパターンに一致するファイルを検索し、一覧をリスト ボックスに表示します。

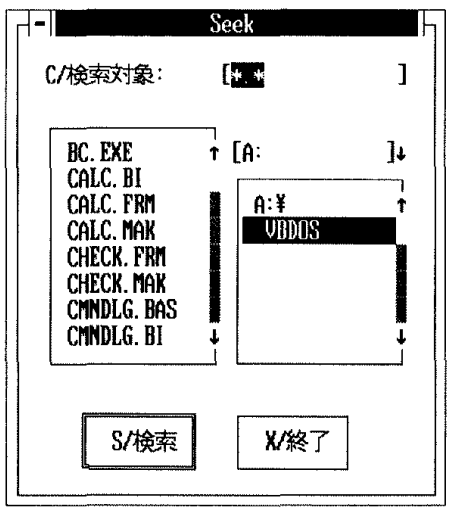


図 10.3　SEEK アプリケーション

SEEK.FRM で使っているコントロールとイベントには、次のようなものがあります。

オブジェクト	プロパティの設定	使用するイベント
フォーム	FormName = frmSeek Caption = Seek	
ラベル	CtlName = lblCriteria Caption = &C/検索対象:	
テキスト ボックス	CtlName = txtSearchSpec Text = *.*	Change, GotFocus
ファイル リスト ボックス	CtlName = filList Pattern = *.*	
ドライブ リスト ボックス	CtlName = drvList	Change
ディレクトリ リスト ボックス	CtlName = dirList	Click, Change
ラベル	CtlName = lblFound Caption = &F/ファイル検索: Visible = False	
ラベル	CtlName = lblCount Caption = Count Visible = False	
リスト ボックス	CtlName = lstFoundFiles Visible = False	
コマンド ボタン	CtlName = cmdSearch Caption = &S/検索 Default = True	Click
コマンド ボタン	CtlName = cmdExit Caption = &X/終了	Click

注 意 ファイルシステム コントロールには Caption プロパティがないため、アクセス キーを割り当てるにはラベル コントロールを使います。ラベル コントロールのこのような利用法については、第 4 章「コントロールの作成と利用」を参照してください。

lstFoundFiles リスト ボックスには検索されたファイルが表示されます。このリスト ボックスとラベル lblFound と lblCount は、デザイン時に Visible プロパティが False (0) に設定されているため、起動時や検索動作中には表示されません。必要なときにだけ、Visible プロパティに True (-1) を設定して表示します。

SEEK のコード

ファイルシステム コントロールを利用するとき、Change イベントが発生する条件を理解しておく必要があります。ドライブ リスト ボックスでは、項目の 1 つをユーザーがマウスでシングル クリックすると、Change イベントが発生します。これに対してディレクトリ リスト ボックスでは、ダブル クリックしないと Change イベントが発生しない点に注意してください。シングル クリックしたときはその項目が反転表示され、ListIndex プロパティの値は変更されますが、Path プロパティの値はそのままです。通常、キーボード操作でリスト ボックスから項目を選択するとき、[ENTER] キーを押します。ただしディレクトリ リスト ボックスでは、[ENTER] キーはシングル クリック

の意味になり、単にその項目が反転表示されるだけです。また、多くのダイアログ ボックスでは、[ENTER] キーはデフォルトのコマンド ボタンを選択する意味に使われます。

SEEK アプリケーションを考えるとき、[ENTER] キーが押されたら、ユーザーはディレクトリを変更したいのか、それとも <検索> ボタンを押した意味でファイルの検索を実行したいのか、どちらであるかを判断する必要があります。SEEK アプリケーションでは、LastChange という名前のグローバル変数をフラグとして使い、この問題を解決しています。具体的には、デフォルトのボタンである <検索> コマンド ボタン (cmdSearch) に対する Click イベントが発生したら、ユーザーが最後に行った操作がディレクトリ リスト ボックスに対するものであるかを判断し、そうであればディレクトリ リスト ボックス (dirList) の Path プロパティを変更するようにします。

SEEK のモジュールレベルでは、次のような定義を行っています。

```
CONST  DIRSBOXCLICK = 1
DIM SHARED LastChange AS INTEGER
```

グローバルな変数 LastChange が、この判断を行うためのフラグです。この変数に、定数 DIRSBOXCLICK として定義されている値 (1) が代入されていれば、新しいディレクトリの選択であると判断し、0 であればデフォルトのボタンである <検索> の選択と解釈します。LastChange は、dirList_Click イベント プロシージャで 1 に設定されます。ユーザーが [ENTER] キーを押すかコマンド ボタン cmdSearch をクリックしたときに、LastChange 変数の値が 1 のままであれば、そのとき反転表示されているディレクトリの名前 (dirList.List(dirList.ListIndex) が filList.Path に代入されます。これによって、ユーザーが方向キーと [ENTER] キーで行った操作を、ディレクトリ リスト ボックス中の項目のダブル クリックと同じ意味にしています。

SEEK アプリケーションに含まれるイベント プロシージャは次のとおりです。

イベント プロシージャ	イベント発生の条件	動作
drvList_Change	ユーザーが、ドライブ リスト ボックスから新しいドライブをクリックするか、反転表示させて [ENTER] キーを押すと発生します。 選択されたドライブは、自動的にドロップダウン リスト ボックスの最上部に表示されます。	新しく選択されたドライブの名前 (drvList.Drive) をディレクトリ リスト ボックスの Path プロパティに代入し、ドライブ リスト ボックスとディレクトリ リスト ボックスの同期を取ります。 また、選択したドライブが無効な場合のエラー処理も行っています。
dirList_Click	ユーザーが、ディレクトリ リスト ボックスから新しいディレクトリをクリックするか (ダブル クリックではありません)、方向キーで別のディレクトリを反転表示させたときに発生します。	変数 LastChange に DIRSBOXCLICK の値 (1) を代入し、ユーザーがディレクトリ リスト ボックスを操作中であることを示します。LastChange の値は、デフォルトのコマンド ボタン (cmdSearch) が選択されたときに調べられます。

dirList_Change	ユーザーが、ディレクトリ リスト ボックスから新しいディレクトリをダブル クリックしたときに発生します。また、ユーザーが方向キーで別のディレクトリ名を反転表示させて [ENTER] キーを押したとき、cmdSearch_Click イベント プロシージャ中でディレクトリ リスト ボックスの Path プロパティが変更されることによっても発生します。ディレクトリ リスト ボックスの表示内容は、新しく選択されたディレクトリを中心としたものに自動的に変更されます。	新しく選択されたディレクトリの名前 (dirList.Path) をファイル リスト ボックスの Path プロパティに代入し、ディレクトリ リスト ボックスとファイル リスト ボックスの同期を取ります。 また，変数 LastChange の値を 0 にリセットし、ユーザーのディレクトリ リスト ボックスに対する操作が終わったことを示します。
cmdSearch_Click	ユーザーが <検索> コマンド ボタンをクリックすると発生します。また、このボタンはデフォルトのコマンド ボタンであるため、[ENTER] キーを押しても発生します。	変数 LastChange の値を調べて、新しいディレクトリの選択であるのか、検索の開始であるのかを判断します。ディレクトリの選択であればディレクトリ リスト ボックスの設定を変更し、検索の開始であれば画面を書き換えてから検索を行います。

drvList_Change

ユーザーが、ドライブ リスト ボックス (drvList) で新しいドライブを選択すると、このイベント プロシージャが実行されます。

```
SUB drvList_Change ()
    'エラー処理ルーチンを設定します。
    ON LOCAL ERROR GOTO DriveHandler
    dirList.Path = drvList.Drive
    EXIT SUB

DriveHandler:
    'エラーならドライブを元に戻します。
    drvList.Drive = dirList.Path
    EXIT SUB
END SUB
```

ドライブ リスト ボックスの Change イベントは、マウスや方向キーによって、新しいドライブが選択されると発生するという点に注意してください。フロッピー ディスクが正しくセットされていないドライブにアクセスしようとしたり、接続されていないネットワーク ドライブを選択しようとすると、エラー処理ルーチンに制御が移ります。この場合、ディレクトリ リスト ボックスの Path プロパティ (ditList.Path) に対する代入は行われず、元の設定のままで残されます。エラー 処理ルーチンでは、この設定をドライブ リスト ボックスに書き戻しています。

dirList_Click

すでに述べたとおり、ディレクトリ リスト ボックスに表示されている項目をシングル クリックしても、Change イベントは発生しません。もし、別の項目が反転表示されただけで Change イベントが発生するようになっていると、ユーザーが方向キーを押すたびに別のディレクトリのチェックが行われ、処理に長い時間がかかってしまうため、ディレクトリ リスト ボックスの Change イベントはダブル クリックによって発生するようになっています。

このイベント プロシージャでは、ユーザーがディレクトリ リスト ボックスを操作していることを示すために、フラグ (LastChange 変数) を 1 (DIRSBOXCLICK) に設定します。それ以外の処理は行われず、ディレクトリの表示も変更されません。

```
SUB dirList_Click ()
    LastChange = DIRSBOXCLICK
END SUB
```

dirList_Change

ユーザーがディレクトリ リスト ボックスの項目をダブル クリックするか、コード中からディレクトリ リスト ボックスの Path プロパティが変更されたときに実行されます。

```
SUB dirList_Change ()
    filList.Path = dirList.Path
    LastChange = 0
END SUB
```

このプロシージャの最初の行では、ディレクトリ リスト ボックスの Path プロパティの値をファイル リスト ボックスの Path プロパティに代入しています。この結果、ファイル リスト ボックスの表示内容は変更されます。変数 LastChange が以前に 1 に設定されていた場合は、0 にリセットされます (例えばユーザーがディレクトリ リスト ボックスのある項目をクリックした後に別の項目をダブル クリックした場合は、LastChange が 1 に設定された状態で dirList_Change が呼び出されます)。

cmdSearch_Click

SEEK アプリケーションでは 2 つのコマンド ボタンを使っていますが、cmdSearch コマンド ボタンは、検索対象を指定する画面の <検索> ボタンに使われています。このボタンを選択すると、実際に検索が始まります。デフォルトのコマンド ボタンに設定されているため、ユーザーが [ENTER] キーを押すと、このイベント プロシージャが呼び出されます。

検索を開始すると、フォーム上のコントロールの配置を変えて、検索の結果見つかったファイル名を表示する画面が表示されます。この状態では、cmdSearch コマンド ボタンの Caption プロパティが変更され、<戻る> ボタンに使われます。

このイベント プロシージャでは、まずどちらの画面が表示されているのかをチェックします。また、ディレクトリ リスト ボックスのダブル クリックの意味の [ENTER] キーであるかどうかをチェックします。

```
SUB cmdSearch_Click ()
    '検索結果表示中の場合は初期画面に戻ります。
    '<戻る>なら、いったんアンロードして再びロードします。
    IF cmdSearch.Caption = "&R/戻る" THEN
        UNLOAD frmSeek
        SHOW
        EXIT SUB
    END IF

    'ディレクトリ リスト ボックス操作中であれば
    'ディレクトリの選択と解釈します。
    IF LastChange = DIRSBOXCLICK THEN
        dirList.Path = dirList.List(dirList.ListIndex)
        EXIT SUB
    END IF

    '指定形式で検索
    'フォームの外観を変更します。
    Visible = FALSE
    Width = 70
    Left = (SCREEN.Width - Width) ¥ 2
    txtSearchSpec.Visible = FALSE
    drvList.Visible = FALSE
    dirList.Visible = FALSE
    filList.Visible = FALSE
    lblCriteria.Visible = FALSE
    cmdSearch.Visible = FALSE
    lblCount.Caption = "0"
    lblFound.Visible = TRUE
    lblCount.Visible = TRUE
    lblCount.Top = 1
    lblCount.Left = 17
    lblFound.Top = 1
    lblFound.Left = 2
    cmdExit.Caption = "取消"
    cmdExit.Left = 28
    lstFoundFiles.Left = 1
    lstFoundFiles.Top = 2
    lstFoundFiles.Width = ScaleWidth - 2
    lstFoundFiles.Visible = TRUE
    Visible = TRUE

    filList.Pattern = txtSearchSpec.Text
    FirstPath$ = dirList.Path
    DirCount% = dirList.ListCount

    '再帰的なディレクトリの検索を始めます。
    NumFiles = 0          'グローバルのファイル検索個数をリセットします。
    result% = DirDriver(FirstPath$, DirCount%, "")

    'フォームを一度アンロードしてから再ロードし、処理をリセットします。
    IF result% = TRUE THEN
        UNLOAD frmSeek
        SHOW
        EXIT SUB
    END IF
```

```
    filList.Path = dirList.Path
    cmdSearch.Caption = "&R/戻る"
    cmdSearch.Left = 22
    cmdExit.Caption = "&X/終了"
    cmdExit.Left = 34
    cmdSearch.Visible = TRUE
    cmdSearch.SETFOCUS
END SUB
```

10.3　コモン ダイアログ ツールキット

Visual Basic for MS-DOS は、ファイルのオープンやテキストの検索など、アプリケーションでよく使うダイアログ ボックスを集めた、コモン ダイアログ ツールキット (CMDDLG) を用意しています。このツールキットには、次の 7 種類のダイアログ ボックスが含まれています。

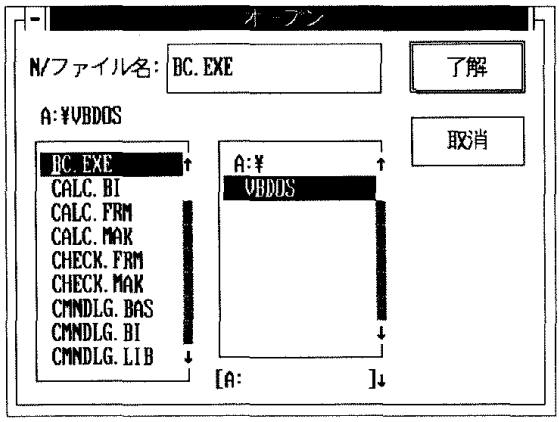
タイトル	用途	呼び出すルーチン
[オープン]	ファイルのオープン	FileOpen
[保管]	ファイルの保管	FileSave
[印刷]	ファイルの印刷	FilePrint
[検索]	文字列の検索	FindText
[置換]	文字列の置き換え	ChangeText
[カラー パレット]	画面の表示色の設定	ColorPalette
[このプログラムについて]	..について	About

これらのダイアログ ボックスの基本的な構成は、Microsoft Windows で使われるものと同じです。コモン ダイアログ ツールキットを使うと、汎用的なダイアログ ボックスを簡単にアプリケーションに組み込むことができます。ツールキットには、Quick ライブラリ (.QLB) やオブジェクト モジュール ライブラリ (.LIB) のほかにソース ファイル (.BAS, .FRM, .BI) も含まれていますから、アプリケーションの仕様に合わせてカスタマイズすることができます。コモン ダイアログ ツールキットに含まれるルーチンを呼び出すために必要な宣言は、インクルード ファイル CMNDLG.BI に含まれています。インクルード ファイルは、[補助] メニューの [パス設定] コマンドで設定したディレクトリになければなりません。.BAS, .FRM, .BI ファイルをアプリケーションに組み込む方法については、第 6 章「プロジェクトの管理」を参照してください。

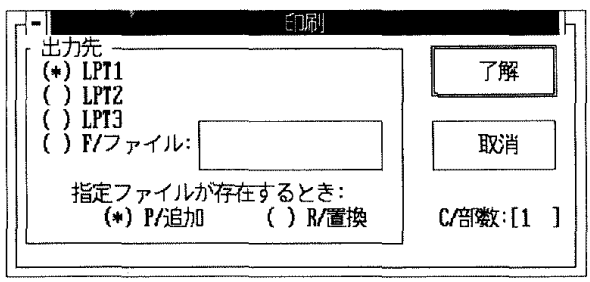
コモン ダイアログ ツールキットのプロジェクト ファイル (CMNDLG.MAK) をプログラミング環境に読み込んで実行すると、このツールキットに含まれているすべてのダイアログ ボックスを表示してみることができます。

フォーム モジュールが 1 つであることからも分かるように、基本的なフォームは 1 つだけです。このフォームの上に、コンテナとして機能している 5 種類のピクチャー ボックスが配置されています。例えば [オープン] ダイアログ ボックス用のピクチャー ボックスは、ファイルの保管用のダイアログ ボックスとしても使われており、必要な初期化を行った後、フォーム上に配置されます。このとき、他のピクチャー ボックスは存在しますが表示されません。

アプリケーションから呼び出すプロシージャは、CMNDLG.BAS に含まれています。例えば [オープン] ダイアログ ボックスを利用するときには、必要な宣言を行った後で、FileOpen **SUB** プロシージャを呼び出します。このプロシージャは、CMNDLGF.FRM に記録されているフォーム上のコントロールを初期化を行い、次のようなダイアログ ボックスを画面に表示します。



例えば、ファイルの印刷のときに使うダイアログ ボックスを呼び出したいときは、FilePrint **SUB** プロシージャを呼び出します。この場合でも表示されるのは同じフォームですが、印刷のダイアログ ボックス用に用意している別のピクチャー ボックスが使われるため、次のようにまったく印象が異なるダイアログ ボックスが表示されます。



コモン ダイアログ ツールキットは比較的大きなサンプルであるため、1 つのダイアログ ボックスをアプリケーションに加えるためにツールキット全体を組み込むと、ファイル サイズが増大するとともに、実行時間も遅くなってしまいます。そこで、コモン ダイアログ ボックスを利用するときは、CMNDLGF.FRM と CMNDLG.BAS を別の名前でコピーし、不要なピクチャー ボックスとプロシージャを削除してから、アプリケーションに加えるようにします。また、CMNDLG.BI を参考に、必要な宣言をアプリケーションに追加してください。

[オープン] ダイアログ ボックス

この章の残りの部分では、コモン ダイアログ ツールキットに含まれている、ファイルの [オープン] ダイアログ ボックスについて説明します。通常は、[ファイル] メニューの [オープン] コマンドをユーザーが選択したときに、このダイアログ ボックスが呼び出されるようにします。

呼び出しの方法

[オープン] ダイアログ ボックスを表示するには、8 つの引数を指定して、FileOpen **SUB** プロシージャを呼び出します。FileOpen **SUB** プロシージャは、CMNDLG.BI で次のように宣言されています。

```
DECLARE SUB FileOpen(FileName AS STRING, PathName AS
 ⇒ STRING, DefaultExt AS STRING, DialogTitle AS STRING,
 ⇒ ForeColor AS INTEGER, BackColor AS INTEGER, Flags
 ⇒ AS INTEGER, Cancel AS INTEGER)
```

アプリケーションから呼び出すときは、呼び出しを行うすべてのモジュールのモジュール レベルにこの宣言を記述するか、CMNDLG.BI をインクルードする必要があります。

最初の 2 つの引数は、呼び出し側が FileOpen **SUB** プロシージャから結果を受け取るための変数で、ダイアログ ボックスに対してユーザーが指定した内容が格納されます。プロシージャの実行が終了したときに、ユーザーが指定したオープンするファイル名とファイルのパス名がそれぞれ格納されて返されます。呼び出し時にこれらの変数に格納されている値は、それぞれファイル名とパス名のデフォルトとしてダイアログ ボックスに表示されます。

その他の引数の意味については、CMNDLG.BAS のコメントを参照してください。

FileOpen の典型的な呼び出し方法を、次に示します。CMNDLG.BAS のモジュールレベルには、ツールキットに含まれるすべてのダイアログ ボックスに対する典型的な呼び出しが記述されていますから参考にしてください。

```
CALL FileOpen (NewFileName$, NewPathName$, "*.txt", "",
 ⇒ 0, 7, Flags%, Cancel%)
```

FileOpen プロシージャ

FileOpen **SUB** プロシージャは、CMNDLG.BAS ファイルに記述されています。このプロシージャは、フォームとコントロールの初期化を行い、フォーム、つまり [オープン] ダイアログ ボックスを表示します。

コントロールとイベント

[オープン] ダイアログ ボックスも、SEEK アプリケーションと同様に、ファイルを操作するための 3 種類のコントロールを組み合わせて使っています。使われているコントロール，主なプロパティの設定，主要なイベントは、次のようになっています。

コントロール	プロパティの設定	主要なイベント
ピクチャー ボックス	CtlName = pctFileOpen	
ラベル	CtlName = lblOpenFile	
テキスト ボックス	CtlName = txtOpenFile	GotFocus
ラベル	CtlName = lblOpenPath	
ファイル リスト ボックス	CtlName =filOpenList	Click, DblClick, PatternChange, PathChange
ドライブ リスト ボックス	CtlName =drvOpenList	Change
ディレクトリ リスト ボックス	CtlName = dirOpenList	Change
コマンド ボタン	CtlName = cmdOpenOK Caption = 了解 Default = True (コード中から設定)	Click
コマンド ボタン	CtlName = cmdOpenCancel Caption = 取消 Cancel = True (コード中から設定)	Click

[オープン] ダイアログ ボックスで使うイベント プロシージャには、次のようなものがあります。これらのプロシージャは、すべて CMNDLGF.FRM ファイルに記述されています。

プロシージャ	呼び出し方法	動作
txtOpenFile_GotFocus	ファイル名またはパターンの入力用のテキスト ボックスにフォーカスが移ると呼び出されます。	テキスト ボックスに表示されているテキストを選択状態にします。
filOpenList_Click	ユーザーがファイル リスト ボックスに表示されている項目をマウスでクリックするか、フォーカスを持っているときに方向キーを押すと呼び出されます。	反転表示しているファイル名をテキスト ボックスに表示します。
filOpenList_DblClick	ユーザーがファイル リスト ボックスに表示されている項目をダブルクリックするか、テキスト ボックスに特定のファイル名が表示されている状態で <了解> ボタンを押したときに呼び出されます。	フォームを非表示にして、<取消> ボタンの Tag プロパティに文字列 "FALSE" を代入します。ファイルの選択が確定したことになり、FileOpen プロシージャに制御が戻ります。
filOpenList_PatternChange	ユーザーがテキスト ボックスに新しいパターンを入力して <了解> ボタンを選ぶと呼び出されます。	パターンを示す文字列を、他との統一を取るために大文字に変換します。
filOpenList_PathChange	ユーザーが新しいドライブやディレクトリを選択したり、テキスト ボックスに新しいパスを入力して <了解> ボタンを選ぶと呼び出されます。	ドライブ リスト ボックスおよびディレクトリ リスト ボックスとの同期を取ります。

drvOpenList_Change	ユーザーがドライブ リスト ボックスから新しいドライブを選択するか、テキスト ボックスに新しいドライブ名を入力して、<了解> ボタンを選ぶと呼び出されます。	ディレクトリ リスト ボックスとの同期を取ります。
dirOpenList_Change	ユーザーがドライブ リスト ボックスまたはディレクトリ リスト ボックスから新しいドライブまたはディレクトリを選択するか、テキスト ボックスに新しいパスを入力して <了解> ボタンを選ぶと呼び出されます。	ファイル リスト ボックスとの同期を取ります。また、選択されているディレクトリを示しているラベルの表示と、テキスト ボックスのパターンの表示を更新します。
cmdOpenOK_Click	ユーザーが <了解> ボタンをクリックするか [ENTER] キーを押すと呼び出されます。このコマンド ボタンの Default プロパティは、FileOpen プロシージャで TRUE に設定されています。	ユーザーがファイル名の指定を確定したのか、単に表示を更新しようとしただけなのかを判断し、それぞれ必要な処理を行います。
cmdOpenCancel_Click	ユーザーが <取消> ボタンをクリックするか、[ESC] キーを押すと呼び出されます。このコマンド ボタンの Cancel プロパティは、FileOpen プロシージャで TRUE に設定されています。	フォームを非表示にします。FileOpen プロシージャに制御が戻ります。

ユーザーは、ドライブ リスト ボックスやディレクトリ リスト ボックスを操作して、各ディレクトリに記録されているファイルを調べることができます。最終的には、オープンする 1 つのファイルの選択を確定するか、操作をキャンセルしてダイアログボックスを閉じるかのどちらかの操作を行います。ユーザーがファイル リスト ボックスに表示されているファイルのどれかをダブル クリックすると、選択が確定します。また、テキスト ボックスに (ワイルド カードを含まない) 特定のファイル名が表示されているときに [ENTER] キーを押したときも、cmnOpenOK_Click イベント プロシージャから filOpenList_DblClick イベント プロシージャが呼び出され、選択が確定します。

cmnOpenOK_Click イベント プロシージャの内容は次のとおりです。

```
SUB cmdOpenOK_Click ()

    '"オープン" と "保管" ダイアログは共通のピクチャー ボックスを使用してい
    'ます。そのピクチャー ボックスに配置されている <了解> ボタンが押された
    'ときの処理です。ユーザーがファイルを選択するか、パスとパターンを変更し
    'た場合は新しいファイル名とパスを設定します。

    'エラー処理ルーチンを設定します。
    ON LOCAL ERROR GOTO OKError

    '<了解> ボタンにフォーカスを移します。
    cmdOpenOK.SETFOCUS
```

```
    'ユーザーが、ディレクトリ リスト ボックスでクリックするか、方向キー
    'を押したときにも、ディレクトリ リスト ボックスのパスが更新される
    'ようにします。(ダブル クリックされたときは自動的に更新されます)
    dirOpenList.Path = dirOpenList.List(dirOpenList.ListIndex)

    'テキスト ボックスに表示されている内容が、ファイル リスト ボックスに
    '表示されている内容と一致していないとき、テキスト ボックスの内容を
    'リスト ボックスの FileName プロパティに設定し、パスやパターンの変更
    'を行います。
    IF filOpenList.FileName <> txtOpenFile.Text THEN
        '現在のパターンを保管します。
        OldPattern$ = filOpenList.Pattern

        'テキスト ボックスの内容をファイル リスト ボックスに設定します。
        filOpenList.FileName = txtOpenFile.Text

        '有効なファイル名が与えられていない (DblClick イベントの後でも
        'ダイアログが表示されている) 場合は、テキスト ボックスの内容を
        '適切なものに置き換えます。
        IF Visible = TRUE THEN
            'テキスト ボックスの表示にワイルド カードが含まれていない
            'ときは、ファイルが見つからないことを示すダイアログ ボックス
            'を表示して、ファイル リスト ボックスの表示を前のパターン
            'に戻します。
            IF (INSTR(txtOpenFile.Text, "*") +
             ⇒ INSTR(txtOpenFile.Text, "?") < 1) THEN
                IF INSTR(Tag, "SAVE") THEN
                    'ファイル保管の場合は DblClick イベント
                    'プロシージャを呼び出します。
                    CALL filOpenList_DblClick
                ELSE
                    MSGBOX "ファイルが見つかりせん。", 0, Caption
                    filOpenList.Pattern = OldPattern$
                    txtOpenFile.SETFOCUS
                END IF
            ELSE
                '入力されたテキストがワイルド カードを含む場合は、テキスト
                'ボックスに現在のパターンを表示し、フォーカスを移します。
                txtOpenFile.Text = filOpenList.Pattern
                txtOpenFile.SETFOCUS
            END IF
        END IF
    ELSE
        'ファイル リスト ボックスのダブル クリックに対するイベント
        'プロシージャを呼び出します。
        CALL filOpenList_DblClick
    END IF

OKErrorReturn:
    EXIT SUB

'エラー処理ルーチン
OKError:

    'エラーメッセージを表示します。
    MSGBOX ERROR$, 0, Caption
```

```
    'テキスト ボックスにフォーカスを移します。
    txtOpenFile.SETFOCUS

    RESUME OKErrorReturn

END SUB
```

ダイアログ ボックスの終了処理

ユーザーの操作によってダイアログ ボックスがクローズされると、制御は FileOpen プロシージャに戻ります。FileOpen プロシージャの最後では、ユーザーがファイルの指定を確定したのか、ダイアログをキャンセルしたのかをチェックしています。具体的には、frmCmnDlg.cmdOpenCancel.Tag を調べます。キャンセルされている場合は、呼び出し側で引数に指定された変数である Cancel パラメータに True (-1) が代入されます。ファイルの指定が確定された場合は Cancel に False (0) が代入され、ユーザーが指定したオープンするファイルのファイル名が FileName パラメータに、またパス名が PathName パラメータに代入されます。呼び出し側では、これらの 3 つの変数の値を参照して、[オープン] ダイアログ ボックスで指定された必要な情報を得ます。

```
    'ユーザーがダイアログを取り消したかどうかを調べ、終了処理を行います。
    IF frmCmnDlg.cmdOpenCancel.Tag <> "FALSE" THEN
        'ユーザーがダイアログを <取消> ボタンで終了しているときは、
        'Cancel フラグを TRUE に設定します。
        Cancel = TRUE
    ELSE
        '<取消> ボタンで終了していない場合は、指定されたファイル名と
        'パス名をパラメータに代入して、呼び出し側に情報を返します。
        Cancel = FALSE
        FileName = frmCmnDlg.txtOpenFile.Text
        PathName = frmCmnDlg.filOpenList.Path
        fmnCmnDlg.cmnOpenCancel.Tag = ""
    END IF

    '呼び出し時に引数でメモリ常駐に指定されているときは、フォームを非表示
    'にします。指定されていないときは、フォームをアンロードします。どちら
    'の処理を行っても、制御は呼び出し側に戻ります。
    IF LEFT$(frmCmnDlg.Tag, 1) = "H" THEN
        frmCmnDlg.pctFileOpen.Visible = FALSE
        frmCmnDlg.HIDE
    ELSE
        UNLOAD frmCmnDlg
    END IF

    EXIT SUB
```

ファイル リスト ボックスの FileName プロパティ

ファイル リスト ボックスの FileName プロパティは、ファイル リスト ボックス中でそのとき選択されているファイル名を示しますが、参照するときと設定するときで扱いが異なります。FileName プロパティの値を参照すると、ファイル リスト ボックス中でそのとき選択されているファイルの、パスを除いたファイル名だけが返されます。一方、FileName プロパティへの代入を行うときは、ファイル名だけでなく、ドライブやディレクトリ パスの指定も合わせて代入することができます。ドライブやディレクト

リ パスの指定を含む文字列を代入すると、Path および Pattern プロパティの値も変更されます。また、代入する文字列によって、DblClick, PathChange, PatternChange の各イベントが発生します。次のように判断されます。

- 文字列の先頭がドライブ名とコロンの組み合わせか、文字列が円記号 (¥) を含む場合は、新しいパスの情報が Path プロパティに代入されて、PathChange イベントが発生します。
- 文字列の中にワイルド カード (* または ?) が含まれている場合は、そのパターンが Pattern プロパティに代入されて、PatternChange イベントが発生します。
- 文字列が実際に存在する特定のファイルを示す場合は、DblClick イベントが発生します。

FileName プロパティに代入する値が無効 (例えば null 文字列など) のときは、ランタイム エラーが発生します。

ファイル リスト ボックスの Pattern プロパティ

ファイル リスト ボックスの Pattern プロパティは、表示するファイル名のパターンを示します。Pattern プロパティのデフォルトの設定は *.* で、選択されているディレクトリ中のすべてのファイルが表示されます。ディレクトリ中の特定のファイルだけを表示したいときは、コード中から Pattern プロパティに特定のパターンを示す文字列 (*.TXT, NEW???.DOC など) を代入します。指定したパターンに一致するファイルがないときは、ファイル リスト ボックスには何も表示されません。また、指定したパターンが無効 (例えば null 文字列など) のときは、ランタイム エラーが発生します。

Pattern プロパティには、セミコロンで区切ることによって、表示するファイルのパターンを複数指定できます。例えば、文字列として *.EXE; *.COM; *.BAT と指定すると、3 種類の拡張子のいずれかを持つファイルが、すべて表示されます。

第 2 部
プログラミング テクニック

11 デスクトップ出力とキーボード入力

この章では、Basic の入出力 (I/O) に関するステートメントや関数を使って、フォームを使わずに画面に文字を表示したり、キーボードから文字を入力する方法について説明します。

Visual Basic で作成するアプリケーションでは、通常はユーザー インターフェイスにフォームとコントロールを使います。しかし、画面に簡単な質問を表示してユーザーに処理を選ばせるときなど、フォームを使わずに処理できるものもあります。また、複雑な計算の実行など、できるだけ多くのメモリを使いたい処理では、必要に応じてフォームをアンロードしたほうがよいでしょう。

このような、フォームを使わない、または使いたくないアプリケーションやルーチンでは、ユーザー インターフェイスに入出力ステートメントや関数を使います。これによって、フォームの制御に必要なメモリが解放され、アプリケーションで効率よくメモリを利用できます。

11.1 デバイスのサポート

MS-DOS では、“ドライバ”と呼ばれるソフトウェアで、キーボードやマウス，ディスプレイ，ディスク ドライブ，通信ポートなどの周辺装置を制御します。Basic には、キーボード (KYBD:)，ディスプレイ (SCRN:)，ディスク ドライブなどの、多くのデバイスを制御するステートメントが用意されています。ただし、マウスのように Basic からは直接制御できないデバイスもあります。

Visual Basic では、フォームが表示されているとき、キーボードやマウスなどの入力デバイスの制御はフォーム エンジンが行います。ユーザーがキーボードやマウスを操作すると、その情報は“イベント”という形でアプリケーションに渡されます。アプリケーションは、イベント プロシージャを使ってこれらのイベントに対応する処理を行います。

このマニュアルの前半では、フォームが表示されているときに、テキスト ボックスやコマンド ボタンなどのコントロールを使って、アプリケーションとユーザーがデータをやり取りする方法について説明してきました、一方、フォームが表示されていない状態では、ユーザーの入力を受け取るには、従来と同じように .BAS モジュールの **INPUT**，**LINE INPUT**，**INPUT$**，**INPUT¥**，**INKEY$** ステートメントを使います。また、Visual Basic for MS-DOS 起動時に /L VBDOS.QLB オプションを指定すると、マウス ツールキット (MOUSE.BAS) を使ってマウス操作を制御できます。

Visual Basic for MS-DOS は、従来の Basic の画面表示に関する機能に対して上位互換性を持っています。フォームが表示されていないときは、従来の Basic で用意されていた

画面表示に関するステートメントや関数をそのまま使うことができます。また、Basic の **LOCATE**, **PRINT** ステートメントや **POS**, **CRSLN** 関数、グラフィックスに関する命令を使って、Visual Basic のデスクトップ (画面全体) にデータやグラフィックスを表示できます。なお、グラフィックスに関しては、オンライン ヘルプまたは『機種依存情報』を参照してください。

注　意　ContorolPanel(7) プロパティを 0 に設定すると、Microsoft Windows の壁紙に似た機能を実現できます。詳細は第 14 章『アプリケーションの外観の変更』を参照してください。

11.2　デスクトップ

デスクトップとは、フォームの背景の領域のことで、従来の Basic で“画面”と呼んでいたものと同じです。Visual Basic では、デスクトップ上にフォームを表示するのが一般的ですが、フォームを表示しないアプリケーションも作成できます。Visual Basic では、**SCREEN** オブジェクトがデスクトップを表します。フォームが表示されていない状態では、この **SCREEN** オブジェクトがデフォルトのオブジェクトとなり、画面に関する操作は **SCREEN** オブジェクト、つまり画面全体に対して行われると考えることができます。一方、フォームが表示されている状態では、従来の Basic でも用意されていた画面操作を行うステートメントや関数のほとんどは実行できません。例えば、フォームが表示されている状態でグラフィックス ステートメントを実行しようとすると、ランタイム エラーが発生します。

PRINT と **CLS** は特殊な例です。**PRINT** および **CLS** キーワードがフォーム モジュール (.FRM ファイル) 中で使われると、それらは **CLS** および **PRINT** メソッドと解釈され、フォームやピクチャー ボックスに対して作用します。同じキーワードがコード モジュール (.BAS ファイル) 中で使われると、それらは従来と同様に **PRINT** および **CLS** ステートメントとして扱われ、画面全体、つまりデスクトップに対して作用します。逆に言えば、**PRINT** および **CLS** をステートメントとして使いたいときは、必ずコード モジュールに含めなければなりません。フォームが表示されていないときに **PRINT** および **CLS** メソッドを使おうとしたり、フォームが表示されているときに **PRINT** および **CLS** ステートメントを使おうとするとエラーになります。**PRINT** および **CLS** メソッドについては、第 7 章『テキストの処理』を参照してください。

デスクトップやフォームに対して出力を行うステートメントやメソッド、およびキーボードからの入力を行うステートメント，関数には、次のようなものがあります。

ステートメント/メソッド	モジュール	フォーム	ターゲット
PRINT ステートメント	.BAS	非表示	デスクトップ
PRINT メソッド	.FRM	表示	フォームまたはピクチャー ボックス
CLS ステートメント	.BAS	非表示	デスクトップ
CLS メソッド	.FRM	表示	フォームまたはピクチャー ボックス
LOCATE ステートメント	.BAS /.FRM	非表示	デスクトップ

VIEW PRINT ステートメント	.BAS /.FRM	非表示	デスクトップ
POS ステートメント	.BAS/.FRM	非表示	デスクトップ
CRSLN ステートメント	.BAS/.FRM	非表示	デスクトップ
INPUT ステートメント	.BAS/.FRM	非表示	デスクトップ
LINE INPUT ステートメント	.BAS/.FRM	非表示	デスクトップ
INPUT$ 関数	.BAS/.FRM	非表示	デスクトップ
INPUT¥ 関数	.BAS/.FRM	非表示	デスクトップ
INKEY$ 関数	.BAS/.FRM	非表示	デスクトップ
WIDTH ステートメント	.BAS/.FRM	非表示	デスクトップ
TAB 関数	.BAS/.FRM	表示および非表示	デスクトップ、フォームまたはピクチャー ボックス
SPC 関数	.BAS/.FRM	表示および非表示	デスクトップ、フォームまたはピクチャー ボックス

11.3 デスクトップへのテキスト表示

ここでは、次のような処理について説明します。

- **PRINT** ステートメントによるデスクトップへのテキストの表示
- **PRINT USING** ステートメントによる書式化されたテキストの表示
- **SPC** 関数によるテキスト表示行のスペースのスキップ
- **TAB** 関数によるテキスト表示行の指定した桁へのスキップ
- **WIDTH** ステートメントによる表示列数と行数の設定
- **VIEW PRINT** ステートメントによるテキスト ビューポートの設定

注 意 MS-DOS では、特に指定しない場合の出力は標準出力デバイスに送られます。デフォルトでは、標準出力デバイスは画面です。MS-DOS のコマンドラインで記号 > または >> を使うと、標準出力をリダイレクトできます。これによって、画面に表示される出力をプリンタなどの別の出力デバイスやファイルに送ることができます (リダイレクトについて詳しくは、MS-DOS のマニュアルを参照してください)。

デスクトップ上のテキストの表示位置の指定

テキストを表示するとき、デスクトップやフォームには“行”と“桁”の格子があるものと考えてください。1 行の高さは表示される 1 文字の高さよりもわずかに高く、1 列の幅は表示される 1 文字の幅よりも少し広くなっています。標準的なテキスト モードでのデスクトップは、幅 80 桁×高さ 25 行で構成されています。図 11.1 に示すように、画面に表示された文字は、格子のそれぞれのマスの中に置かれます。1 つのマスは、行数と桁数で指定します。

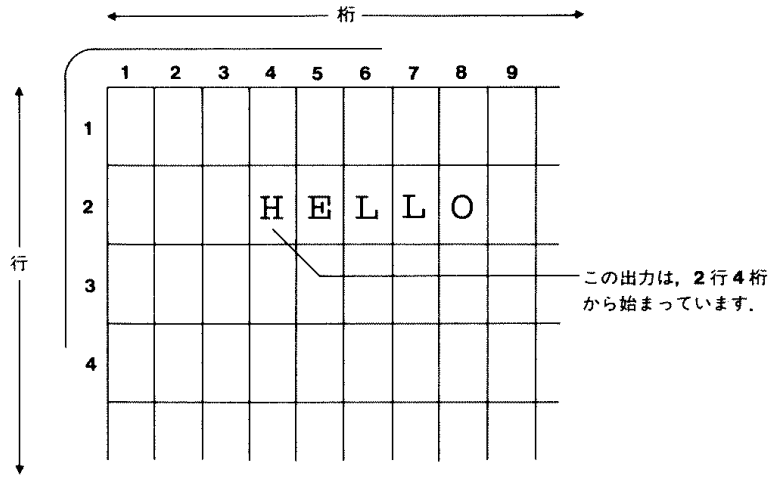


図 11.1　画面へのテキストの出力

テキストの表示位置を指定するには、**LOCATE** ステートメントを使います。**LOCATE** ステートメントについては、この章の「テキスト カーソルの制御」を参照してください。

LOCATE ステートメントでは、デスクトップの左上角が (1,1) になることに注意してください。これに対し、フォーム上の表示位置を指定するときは、左上角は (0,0) になります。

PRINT ステートメントによるテキストと数値の表示

通常、デスクトップへの出力には **PRINT** ステートメントを使います。**PRINT** ステートメントを使うと、数値や文字列を組み合わせてデスクトップに表示できます。また、**PRINT** に何も引数を指定しないと、改行コード (キャリッジ リターンとライン フィード) だけが出力されます。フォームやピクチャー ボックスにテキストを表示するときは、**PRINT** メソッドを使います。**PRINT** メソッドについては、第 7 章「テキストの処理」を参照してください。

PRINT ステートメントの一般的な機能は、次のとおりです。

- **PRINT** ステートメントは、数値の後に必ずスペースを出力します。数値が正のときは、その数値の前にもスペースが挿入されます。数値が負のときは、その数値の前にマイナス記号が付きます。

- 1つの **PRINT** ステートメントで、複数の数式または文字列式の値を表示できます。式は、カンマ (,) または セミコロン (;) で区切ります。デスクトップの各行は、14桁ごとの表示領域に区切られています。カンマを指定すると、次の出力位置は次の表示領域の先頭になります。セミコロンを指定すると、出力位置は移動せず、指定した値は続けて表示されます。プログラミング環境では、式の区切りとして1つ以上のスペースまたはタブを指定してもかまいません。これらはセミコロンと見なされ、[ENTER] キーを押すとセミコロンに変わります。
- **PRINT** ステートメントの最後にカンマまたはセミコロンを指定しないと、最後に改行コード (キャリッジ リターンとライン フィード) が出力されます。最後にカンマを指定すると、次の出力位置は次の出力領域の先頭になり、セミコロンを指定すると、次の出力位置は最後の出力のすぐ後になります。
- 表示するデータが画面の右端を超えるときは、残りは次の行に表示されます。例えば、80桁の画面の左端から100文字のテキストを表示すると、最初の80文字が1行目に表示され、残りの20文字は次の行に表示されます。直前の **PRINT** ステートメントの最後にカンマやセミコロンが指定されていたときなど、そのときの出力位置が行の途中にある場合は、右端になるまで1行目に表示され、残りは次の行の1桁目から表示されます。

次のプログラムは、**PRINT** ステートメントを使った出力例です。

```
A% = 2
B% = -1
C% = 3
X$ = "over"
Y$ = "there"

PRINT A%, B%, C%
PRINT B%, A%, C%
PRINT A%; B%; C%
PRINT X$; Y$
PRINT X$, Y$;
PRINT A%, B%
PRINT
FOR I = 1 TO 7
    PRINT X$,
NEXT
```

出力

```
 2             -1             3
-1              2             3
 2 -1  3
overthere
over           there 2       -1

over            over           over           over
over            over           over
```

PRINT USING による出力の書式指定

PRINT USING は、出力データ、特に数値データを表示するときに、**USING** キーワードを指定しない単独の **PRINT** ステートメントよりも細かく書式を指定できます。**PRINT USING** では、書式指定文字を使って、表示される数値の桁数 (または文字列の文字数) や、数値の前にプラス記号 (+) や円マーク (¥) を表示するかどうかなどを指定できます。

次の例は、**PRINT USING** の使い方を示したものです。書式指定文字列の後に、式を指定しています。**USING** を指定しない **PRINT** と異なり、式の間は必ずセミコロンで区切ります。また、書式指定文字列と最初の式の間もセミコロンで区切ります。

```
X = 441.2318

PRINT USING "小数点以下第 3 位までの数値 ###.###";X
PRINT USING "円記号の付いた数値 ¥¥##.##";X
PRINT USING "指数形式の数値 #.###^^^^";X
PRINT USING "プラス記号の付いた数値 +###  "; X; 99.9
```

出力

```
小数点以下第 3 位までの数値 441.232
円記号の付いた数値 ¥441.23
指数形式の数値 0.441E+03
プラス記号の付いた数値 +441   プラス記号の付いた数値 +100
```

最後の例では、書式指定文字列の後に 2 つの式が指定されていて、両方の式に "プラス記号の付いた数値" という文字列が表示されていることに注意してください。また、# や ^ などの書式指定に使う文字をそのまま表示したいときは、それらの文字の前にアンダーバー (_) を付けてください。

FORMAT$** 関数を使って、出力書式を指定することもできます。**FORMAT$ 関数は、指定した書式に従って数値を文字列に変換します。**PRINT** ステートメントで指定する式の中で **FORMAT$** 関数を使うことができます。

スペースによるスキップと任意の桁への移動

PRINT ステートメントや **PRINT** メソッドで **SPC(*n*)** 関数を使うと、次の例のように *n* 個の半角スペースを表示して、出力位置を制御できます。

```
PRINT "         1         2         3"
PRINT "1234567890123456789012345678 90"
PRINT "First Name"; SPC(10); "Last Name"
```

出力

```
         1         2         3
1234567890123456789012345678 90
First Name          Last Name
```

また、**PRINT** ステートメントや **PRINT** メソッドで **TAB(*n*)** 関数を使うと、その行の (画面の左端から) *n* 番目の桁までスキップできます。次の例は、**TAB** 関数を使って上の例と同じ出力を行います。

```
PRINT "          1         2         3"
PRINT "123456789012345678901234567890"
PRINT "First Name"; TAB(21); "Last Name"
```

SPC および **TAB** 関数を単独で使って画面の表示位置を変えることはできません。これらのステートメントは、必ず **PRINT** ステートメントおよびメソッドや、**LPRINT**, **PRINT #** ステートメントの中で使います。

テキスト ビューポートの設定

これまでは、デスクトップ全体に対してテキストを出力していました。ここでは、**VIEW PRINT** ステートメントを使い、画面を水平に分割した“テキスト ビューポート”によって、出力位置を制限する方法を説明します。**VIEW PRINT** ステートメントの構文は、次のようになります。

VIEW PRINT [*toprow%* **TO** *bottomrow%*]

引数 *topline%* と *bottomline%* には、テキスト ビューポートの上端と下端の行位置を指定します。テキスト ビューポートでは、デスクトップの幅全体が使われることに注意してください。

テキスト ビューポートを指定すると、画面のスクロール範囲を制御できます。ビューポートが設定されていないと、表示出力が画面の最下行に達したとき、画面の一番上に表示されていたテキストは、スクロールされて画面から消えます。**VIEW PRINT** ステートメントを使ってテキスト ビューポートを指定すると、画面の上部または下部に表示した何行かのテキストはそのままで、表示出力はビューポートの範囲内でスクロールします。また、**CLS 2** ステートメントを使うと、テキスト ビューポート内だけを消去し、画面の残りの部分をそのままにすることができます。グラフィックス出力のためのビューポートの設定方法については、『機種依存情報』を参照してください。

フォームが表示されているときは、**VIEW PRINT** ステートメントは使えません。また、ビューポートを指定しているとき、**LOCATE** ステートメントでは必ずビューポート内の位置を指定します。ビューポートの外側を指定すると、「引数が有効範囲にありません」というエラー メッセージが表示されます。

次の例の出力から、**VIEW PRINT** ステートメントの効果が理解できます。

```
CLS
LOCATE 3, 1
PRINT "これはテキストビューポートの上にあります．スクロールされません．"

LOCATE 4, 1
PRINT STRING$(60, "_")          'テキスト ビューポートの上端と
LOCATE 11, 1
PRINT STRING$(60, "_")          '下端に水平線を引きます。

PRINT "これはテキストビューポートの下にあります．"

VIEW PRINT 5 TO 10              'テキスト ビューポートの範囲を 5 行
                                'から 10 行までに設定します。

FOR I% = 1 TO 20                'テキスト ビューポートに数値と
```

```
                                        'テキストを表示します。
    PRINT I%; "行目のテキスト"
NEXT

DO: LOOP WHILE INKEY$ = ""              'キーが押されるのを待ちます。
CLS 2                                   'テキスト ビューポート内だけを
                                        'クリアします。
END
```

出力 (ユーザーがキーを押す前)

```
これはテキストビューポートの上にあります。スクロールされません。
--------------------------------------------------------------
 16 行目のテキスト
 17 行目のテキスト
 18 行目のテキスト
 19 行目のテキスト
 20 行目のテキスト

--------------------------------------------------------------
これはテキストビューポートの下にあります。
```

出力 (ユーザーがキーを押した後)

```
これはテキストビューポートの上にあります。スクロールされません。
--------------------------------------------------------------

--------------------------------------------------------------
これはテキストビューポートの下にあります。

何かキーを押してください
```

11.4 フォームを使ったアプリケーションでのデスクトップへのテキスト表示

フォームを使ったアプリケーションでデスクトップにテキストを表示するときには、必ずすべてのフォームを非表示にしてから行います。通常は、SCREEN.HIDE を使います。フォームを非表示にした後、デスクトップに対して出力を行う処理は、通常はコード モジュールに記述します。非表示にしたフォームを再び表示させるとき、デスクトップに対して行った表示をそのままにして、その上にフォームを表示するか、消去してからフォームを表示するかを選択することができます。デスクトップに対する表示を残したままフォームを表示したいときは、フォームを表示する前に **SCREEN** オブジェクトの ControlPanel (7) プロパティを False (0) (SCREEN.ControlPanel(7)=0) に設定してください。

次の例では、テキスト ボックスに入力したテキストをデスクトップ上に表示しています。テキストを入力してからフォームをダブルクリックするとフォームが消え、テキストがデスクトップ上に表示されます。テキストを表示すると、再びフォームが表示されます。

▼ **フォームを使ったアプリケーションでデスクトップにテキストを表示するには**

1. フォーム デザイナで Text1 というテキスト ボックスを作成します。
2. プログラミング環境の [ﾌｧｲﾙ] メニューの [新規モジュール] コマンドを選び、SHOWTHIS.BAS という名前のコード モジュールを作成します。
3. SHOWTHIS.BAS に次のプロシージャを記述します。

```
SUB ShowMe (ToShow$, Row%, Col%)
    LOCATE Row%, Col%          '出力位置を設定する。
    PRINT ToShow$              'テキストを表示する。
    SLEEP(3)                   '3 秒間待つ。
END SUB
```

4. フォーム モジュールに次のイベント プロシージャを記述します。

```
SUB FORM_DblClick
    SCREEN.HIDE                'フォームを非表示にする。
    CALL ShowMe((Text1.Text), (Text1.Top), (Text1.Left))
    SCREEN.ControlPanel (7) = 0      'デスクトップの表示を残す。
    SCREEN.SHOW                      'フォームを再表示する。
END SUB
```

11.5 キーボードからの入力

ここでは、次のステートメントや関数を使ってキーボードから入力されたデータを受け取る方法について説明します。

- **INPUT** ステートメント
- **LINE INPUT** ステートメント
- **INPUT$** 関数と **INPUT¥** 関数
- **INKEY$** 関数

これらのステートメントや関数は、フォームが表示されているときは利用できません。フォームが表示されている状態では、**INPUTBOX$** 関数が便利です。**INPUTBOX$** 関数については、第 5 章「ダイアログ ボックスの作成」を参照してください。

注　意　MS-DOS では、特に指定しない場合の入力は標準入力デバイスから行われます。デフォルトでは、標準入力デバイスはキーボードです。MS-DOS のコマンドラインで記号 < を使うと、標準入力をリダイレクトできます (リダイレクトについて詳しくは、MS-DOS のマニュアルを参照してください)。

INPUT ステートメント

INPUT ステートメントは、ユーザーが入力したデータを受け取って、そのデータを変数に格納します。

```
INPUT A%, B!, C$
INPUT D$
PRINT A%, B!, C$, D$
```

出力

```
? 6.6,45,a string
? "D$ に入る文字列"
 7             45            a string      D$ に入る文字列
```

INPUT ステートメントの一般的な機能は、次のとおりです。

- **INPUT** ステートメントは、疑問符 (?) とその後ろに点滅するカーソルを表示して、ユーザーに対して入力を求めます。
- **INPUT** ステートメントの後ろには、1 つ以上の変数名を指定します。複数の変数を指定するときは、変数名をカンマで区切ります。
- **INPUT** のプロンプト (疑問符と点滅するカーソル) に対してユーザーが入力する値の個数は、**INPUT** ステートメントで指定された変数の個数と同じでなければなりません。
- ユーザーが入力する値は、**INPUT** ステートメントで指定された変数の型と同じでなければなりません。例えば、変数に整数，長整数，単精度実数，倍精度実数、通貨型のいずれかの型が指定されているときは、数値を入力しなければなりません。また、文字列変数が指定されているときは、文字列を入力しなければなりません。
- 複数の値を入力するときは、カンマで区切らなければなりません。1 つ以上のカンマを含む文字列を入力するときは、ダブル クォーテーション (") で囲みます。ダブル クォーテーションを使うと、文字列が複数の値として扱われるのを防ぐことができます。

入力が上の最後の 3 つの規則に反するときは、「再入力して下さい」というメッセージが表示されます。このメッセージは、入力されたデータが変数の個数と型に一致するまで表示されます。

INPUT ステートメントでは、次の例のようにプロンプトとしてメッセージを表示できます。これにより、ユーザーに入力の内容を分かりやすく知らせることができます。

```
INPUT "正しい時刻 (時, 分) を入力してください"; Hr$, Min$
```

このステートメントを実行すると、次のプロンプトが表示されます。

```
正しい時刻 (時, 分) を入力してください?
```

プロンプト メッセージの指定と変数の間がセミコロンで区切られていることに注意してください。セミコロンを付けると、プロンプト メッセージの後ろに疑問符が表示されます。疑問符を表示したくないときは、次のようにプロンプト メッセージの指定と変数をカンマで区切ってください。

```
INPUT "正しい時刻 (時, 分) を入力してください", Hr$, Min$
```

このステートメントを実行すると、次のプロンプトが表示されます。

```
正しい時刻 (時, 分) を入力してください
```

LINE INPUT ステートメント

LINE INPUT ステートメントは、カンマを含むテキストやスペースで始まるテキスト, スペースで終わるテキストを、ダブル クォーテーションで囲まずに入力したいときに使います。**LINE INPUT** ステートメントは、その名前からもわかるように、キーボードから入力された行 ([ENTER] キーを押すまでのキー入力) を受け取って、1 つの文字列変数に格納します。**LINE INPUT** ステートメントは、**INPUT** ステートメントと異なり、入力のときに疑問符を表示しません。ただし、**INPUT** ステートメントと同様に、プロンプト文字列を表示できます。

次の例は、**INPUT** ステートメントと **LINE INPUT** ステートメントの違いを示します。

```
'入力を異なる 3 つの変数に代入します。
INPUT "3 つの値をカンマで区切って入力してください: ", A$, B$, C$

'入力を 1 つの変数に代入します (カンマは入力の区切りとして扱われません)。
LINE INPUT "同じ 3 つの値を入力してください: ", D$
PRINT "A$ = "; A$
PRINT "B$ = "; B$
PRINT "C$ = "; C$
PRINT "D$ = "; D$
```

出力

```
3 つの値をカンマで区切って入力してください: by land, air, and sea
同じ 3 つの値を入力してください: by land, air, and sea
A$ = by land
B$ = air
C$ = and sea
D$ = by land, air, and sea
```

INPUT ステートメントと **LINE INPUT** ステートメントは、どちらもユーザーが [ENTER] キーを押したときに入力が終了し、カーソルが次の行に移動します。ただし、**INPUT** キーワードとプロンプト メッセージの指定の間にセミコロンを入れると、カーソルは同じ行に表示されます。

```
INPUT "１番目の値：", A
INPUT; "２番目の値：", B
INPUT "　　３番目の値：", C
```

次の出力は、上のプログラムよる入力例とプロンプトの表示位置を示します。

```
１番目の値：5
２番目の値：4　　３番目の値：3
```

INPUT$ 関数，INPUT¥ 関数

INPUT ステートメントと **LINE INPUT** ステートメントは、どちらも [ENTER] キーが押されるまで、入力された値を変数に格納するのを待ちます。これらのステートメントは、入力された行を読み込んで、それを変数に代入します。これに対し、**INPUT$** 関数と **INPUT¥** 関数は、[ENTER] キーの入力を待たずに、指定された数だけ文字を読み込むと変数に格納します。**INPUT$** 関数にはバイト数を、**INPUT¥** 関数には文字数を指定します。次の例では、ユーザーが入力した 3 文字の文字列を読み込み、変数 Test$ に格納します。

```
Test$ = INPUT$(3)
Kanji$ = INPUT¥(3)
```

INPUT$ 関数と **INPUT¥** 関数は、**INPUT** ステートメントとは異なり、プロンプトや入力された文字を画面に表示しません。また、**INPUT$** と **INPUT¥** は関数なので、次のように式の中で使います。

```
INPUT X                    'INPUT はステートメントです。

PRINT INPUT$(1)            'INPUT$ は関数なので、式の中で使います.
Y$ = INPUT$(1)
```

INPUT$ 関数と **INPUT¥** 関数は、キーボードからの入力を、書式指定されていない文字列として読み込みます。**INPUT** ステートメントや **LINE INPUT** ステートメントと異なり、[ESC] や [BS] などの特殊キーを含む任意のキー入力を受け付けます。例えば、次の例で [ENTER] キーが 5 回押されると、5 つの改行文字が変数 Test$ に代入されます。

```
Test$ = INPUT$(5)
```

INKEY$ 関数

キーボードからの入力を処理するための関数に **INKEY$** 関数があります。Basic は **INKEY$** 関数を含む式を見つけると、次のいずれかの条件のあとに、ユーザーがキーを押したかどうかをチェックします。

- 最後に **INKEY$** を含む式を見つけたとき
- アプリケーションを起動した後 (これが最初に見つかった **INKEY$** だったとき)

最後にチェックしてから (またはアプリケーションを起動してから) 何もキーが押されていないとき、**INKEY$** は null 文字列 ("") を返します。いずれかのキーが押されていれば、**INKEY$** はそのキーに対応する文字を返します。

INKEY$** 関数は、ファンクションキーや [INS] キーなどの特殊キーが押されたかどうかをチェックするときにも使えます。次の例では、何かキーが押されたときに、**INKEY$ 関数で入力された文字コードとその長さ (バイト数) を表示します。ファンクション キーや漢字などを入力してみてください。返される値については、『機種依存情報』を参照してください。

```
CLS
DO
    DO
        a$ = INKEY$
    LOOP UNTIL a$ <> ""

    SELECT CASE ASC(a$)
        CASE IS >= &H100
            PRINT "漢字連続コード:";
            PRINT "&H" + HEX$(ASC(a$)) + " ";
            PRINT TAB(40); "<" + a$ + ">";
        CASE 0
            PRINT "拡張キー コード:";
            PRINT "&H0+&H" + HEX$(ASC(MID$(a$, 2, 1)));
            PRINT TAB(40); "----";
        CASE IS <= &H20
            PRINT "1 バイト制御コード:";
            PRINT "&H" + HEX$(ASC(a$)) + " ";
            PRINT TAB(40); "----";
        CASE ELSE
            PRINT "ASCII コード:";
            PRINT "&H" + HEX$(ASC(a$)) + " ";
            PRINT TAB(40); "<" + a$ + ">";
    END SELECT

    PRINT USING "バイト数 = ####        "; LEN(a$)

LOOP UNTIL a$ = CHR$(&H1B)            '[ESC] キーで終了します。
```

INKEY$** 関数は、入力データをチェックしながら他の処理を実行できます。これに対し、**LINE INPUT, INPUT$, INPUT は、入力が終わるまでアプリケーションの実行を中断します。次の例を参照してください。

```
PRINT "任意のキーを押すと開始します。もう 1 度キーを押すと終了します。"

'ユーザーがキーを押すまで、何も実行しません。
Begin$ = INPUT$(1)

I& = 1

'1 から 100 万までの数を表示します。
'ループの実行中、キー入力をチェックします。
DO
    PRINT I&
    I& = I& + 1

'変数 I& の値が 100 万を超えるか、いずれかのキーが押されるまでループします。
LOOP UNTIL I& > 1000000 OR INKEY$ <> ""
```

テキスト カーソルの制御

テキスト カーソルは、アプリケーションからの出力やユーザーからの入力が次に表示されるデスクトップ上の位置を示します。次の例では、**INPUT** ステートメントが "First name: " という 12 文字のプロンプトを表示したあと、カーソルは 1 行 13 桁の位置で入力を待ちます。

```
'画面をクリアし、1 行 1 桁から表示します.
CLS
INPUT "First name: ", FirstName$
```

次の例では、2 番目の **PRINT** ステートメントの終わりにセミコロンが付いているので、カーソルは 2 行 27 桁に表示されます。

```
CLS
PRINT
'次の行には 26 文字あります。
PRINT "Press any key to continue.";
PRINT INPUT$(1)                    '実行を中断し、 キー入力を待ちます。
```

ここでは、デスクトップ上のテキスト カーソルの表示位置の指定，カーソルの形状の変更，カーソル位置の情報の取得について説明します。

注　意　上の例や次の例は、フォームが表示されているときは無効です。また、これらのステートメントでは、テキスト ボックスの挿入ポインタの位置や形状は制御できません。

カーソル位置の指定

デスクトップ上の出力位置を制御するために、**PRINT** ステートメントでカンマやセミコロンを使ったり、**SPC** 関数や **TAB** 関数を使う以外に、**LOCATE** ステートメントを使うと便利です。

LOCATE ステートメントの構文は、次のようになります。

LOCATE [*row%*][, [*column%*][, [*cursor%*][, *start%*[,*stop%*]]]]

LOCATE ステートメントを使うと、デスクトップ上の任意の行または桁にカーソルを移動できます。次の例を参照してください。

```
CLS
FOR Row% = 9 TO 1 STEP 2
    Column% = 2 * Row%
    LOCATE Row%, Column%
    PRINT "12345678901234567890";
NEXT
```

出力

```
12345678901234567890

    12345678901234567890

        12345678901234567890

            12345678901234567890

                12345678901234567890
```

カーソルの形状の変更

LOCATE ステートメントの省略可能な引数 *cursor%, start%, stop%* を使うと、カーソルの形状を変更したり、カーソルの表示/非表示を制御できます。*cursor%* の値を 1 に設定するとカーソルは表示され、0 にするとカーソルは非表示になります。*start%* と *stop%* は、表示されているカーソルの高さを制御します。これらの値は、カーソルを構成するドットの上端と下端を指定します（指定できる値の範囲は、機種により異なります）。

LOCATE ステートメントでは、カーソルの位置を移動せずに、カーソルの表示/非表示や形状の変更だけを行うこともできます。例えば次のステートメントは、カーソルを文字の高さよりも低くしますが、**PRINT** ステートメントや **INPUT** ステートメントのためのカーソルの表示位置は実行前と同じです。

```
LOCATE , , 1, 2, 5        '引数 raw と column はどちらも省略可能です。
```

カーソル位置情報の取得

テキスト ボックスなどのコントロールを使うときは、SelStart プロパティの値をチェックしてテキストの入力位置を簡単に調べることができます。デスクトップのカーソル位置を調べるときは、**CSRLIN** 関数と **POS** 関数を使います。

これらの関数は、**LOCATE** ステートメントと反対の働きをします。**LOCATE** がカーソルを移動させるのに対し、**CSRLIN** と **POS** 関数はデスクトップ上の現在のカーソル位置を戻り値として返します。**CSRLIN** 関数は現在カーソルが位置する行を返し、**POS** 関数は現在カーソルが位置する桁を返します。

POS 関数の引数 *numeric-expression* は ダミーの引数で、意味を持ちません。通常は 0 を指定します。

次の例では、**POS** 関数を使って、50 個のアスタリスクを 1 行に 13 個ずつ表示します。

```
LOCATE 1,1
FOR I% = 1 TO 50
    PRINT "*";        'アスタリスクを 1 つ表示し、カーソルの位置は
                      'そのままにします。
    IF POS(0) > 13 THEN PRINT        'カーソル位置が 13 桁を超えると、
                                     'カーソルを次の行に移動します。
NEXT
```

出力

```
*************
*************
*************
***********
```

12 ファイルとデバイスの操作

Visual Basic の入出力 (I/O) ステートメントと関数を使うと、ファイルのデータの読み書きやプリンタへの出力，シリアル ポートを使った周辺装置との通信を行うことができます。

12.1 データ ファイルの操作

プログラムで扱われる情報を“データ”と呼びます。“データ ファイル”は、情報を記録し、名前を付けてディスクに保管したものです。データ ファイルを利用すると、次のようなことを実行できます。

- 大量のデータを作成，操作，保管する。
- 1 つのプログラムから複数のデータへアクセスする。
- 複数のプログラムで同じデータを利用する。

データ ファイルの構成

多くのデータ ファイルは、“レコード”と呼ばれる、関連する情報ブロックの集まりから構成されています。データ ファイル中のレコードは、さらにいくつかの“フィールド”に分かれています。フィールドは、データ ファイルを構成する最小の単位になります。

例えば、ある会社の求職者の履歴書をまとめたファイルは、1 つのデータ ファイルとして考えることができます。ここで、それぞれの人の履歴書がレコードに当たります。さらに、履歴書に記述されている名前や生年月日などの各項目がフィールドに当たります。

注　意　ファイルをレコード単位に扱うのではなく、書式化されていないバイトの列として扱いたいときは、この章の「バイナリ ファイルの I/O」を参照してください。

シーケンシャル アクセス ファイルと ランダム アクセス ファイル

Basic で扱うファイルは、データの保管やアクセスの方法によって、“シーケンシャル アクセス ファイル” と “ランダム アクセス ファイル” の 2 つに分けられます。

シーケンシャル アクセス ファイルは、カセット テープに例えることができます。カセット テープ上のある曲を探すときは、探している曲が見つかるまでテープを先頭から早送りしなければならず、すぐに見つけることができません。シーケンシャル アクセス ファイルの中の情報を探すときも、これと同様です。500 番目のレコードを探すときは、まず 1 番目から 499 番目までのレコードを読み出すことになります。

これに対し、ランダム アクセス ファイルは、CD (コンパクトディスク) に例えることができます。CD 上のある曲を聴きたいときは、CD プレイヤーに何番目の曲と指定するだけです。聴きたい曲の前にある曲をすべて聴かなくても、任意の曲をランダムに呼び出すことができます。同様に、ランダム アクセス ファイルでは、レコード番号を指定するだけでファイルの中のすべてのレコードに直接アクセスでき、アクセス時間を大幅に短縮できます。

ファイル アクセスの種類

データ ファイルへのアクセス方法には、次の 3 種類があります。

- シーケンシャル アクセス
- ランダム アクセス
- バイナリ アクセス

Basic では、これらのアクセス方法を状況に応じて使い分けます。

次の表は、ファイル アクセスの種類，おもに使うファイル，典型的なステートメントを示しています。

ファイル アクセスの種類	おもに使うファイル	典型的なステートメント
シーケンシャル アクセス	テキスト ファイルなどのシーケンシャル アクセス ファイル	**OPEN, LINE INPUT #, PRINT #, INPUT$, CLOSE INPUT #, WRITE #**
	可変長レコードで構成されたファイル (ランダム アクセス ファイル)	
ランダム アクセス	固定長レコードで構成されたファイル (ランダム アクセス ファイル)	**TYPE...END TYPE, OPEN, CLOSE, PUT, LEN, GET**
バイナリ アクセス	すべてのファイル (ファイルは書式化されていないバイトの並びとして扱われる)	**OPEN, CLOSE, SEEK, INPUT$**

“シーケンシャル アクセス”は、テキスト ファイル (MS-DOS の TYPE コマンドで、通常のテキストとして表示されるファイル) などにアクセスするときに使います。テキスト ファイルでは、それぞれのバイトは文字を表すと見なされます。シーケンシャル アクセスを使って作成されるファイルでは、数値は文字列として表されます。シーケンシャル アクセスでは、テキスト エディタを使って作成されるファイルの読み込みや編集を簡単に実行できます。また、テキスト ファイルではレコードの概念は重要ではありません。

“ランダム アクセス”は、固定長のレコードで構成されたファイル (ランダム アクセス ファイル) を扱うときに使います。このようなファイルに対して TYPE コマンドを実行すると、文字列データは通常のテキストとして表示されますが、数値は文字列ではなくバイナリ形式で格納されるので、通常は表示が乱れます。

“バイナリ アクセス”では、ランダム アクセスやシーケンシャル アクセスと同じステートメントを使いますが、データは単なるバイトの並びとして処理されます。したがって、テキスト ファイルやレコードに区切られているファイル，実行可能ファイルなど、どのようなファイルも扱うことができます。

ファイル アクセスの種類に関係なく、**FREEFILE, EOF, LOF** の各関数を使うと、利用できるファイル番号，ファイルの最後，ファイルの長さを調べることができます。また、**SEEK, GET, PUT, INPUT$** などのステートメントは、さまざまなファイルに適用できます。

シーケンシャル アクセスを使って、固定長のレコードで構成されるファイルを処理することもできます。以前の Basic ではランダム アクセスをサポートしておらず、**WRITE #, INPUT #, FIELD, LSET, RSET** などのステートメントを使って、シーケンシャル アクセスでレコードを扱っていました。この方法では、ランダム アクセスを使った場合よりも処理が複雑で効率も低下するため、新規に作成するアプリケーションにはおすすめできません。なお、これらのステートメントは Visual Basic でもサポートされています。フィールドの区切りにカンマを使っているシーケンシャル アクセス ファイルでは、これらのステートメントを使うこともできます。**WRITE#** と **INPUT#** についての詳しくは、「リファレンス」またはオンライン ヘルプを参照してください。

ファイル アクセスの概要

ここでは、ファイルにアクセスする一般的な手順を説明します。この手順は、ファイル アクセスの種類にかかわらず共通です。

1. **OPEN** ステートメントでファイルをオープンし、ファイルに番号を割り当てます。この番号をファイル番号と呼び、ファイルをオープンしている間は、このファイル番号によって操作対象のファイルを指定します。コード中では、番号の前に # 記号を付けて指定します。

   ファイルをオープンするときは、ファイルへのアクセス方法に応じて、次の 5 つのモードのいずれかを指定します。

   - シーケンシャル **INPUT** モード
   - シーケンシャル **OUTPUT** モード

- シーケンシャル **APPEND** モード
- **RANDOM** アクセス モード
- **BINARY** アクセス モード

2. ファイルのすべてまたは一部のデータを読み込み、変数に格納します。

   新規にデータ ファイルを作成したり、データを追加するだけの場合は、当然この処理は必要ありません。シーケンシャル アクセスでは、ファイルの最後に達するか、目的のデータが見つかるまで、ファイルの先頭から順番にデータを読み込みます。ランダム アクセスでは、レコード番号を使ってファイル中の特定のレコードにアクセスします。バイナリ アクセスでは、ファイル中のそれぞれのバイト位置にアクセスできます。

3. 変数に格納されたデータに対して、必要な処理を行います。

4. 新しいデータまたは修正したデータでファイル全体を上書きするか、ファイル中の特定の部分を更新して、ファイルの内容を変更します。

   シーケンシャル アクセスでは、ファイル全体を上書きするか、新しいデータを最後に追加します。ランダム アクセスでは、レコード単位の変更を行います。バイナリ アクセスでは、ファイル中のバイト位置を指定し、バイト (またはそのバイトに続くバイト列) 単位で変更を行います。

5. ファイルに対する処理が終わったら、**CLOSE** ステートメントでファイルをクローズします (シーケンシャル アクセス ファイルのアクセス モードを変更するときにも、**CLOSE** ステートメントで 1 度ファイルをクローズしてから、再度別のモードでオープンする必要があります)。

データ ファイルのオープン

プログラムでデータ ファイルの読み込み，変更，追加を行うときは、まず最初にファイルをオープンしなければなりません。ファイルのオープンには、**OPEN** ステートメントを使います。また、**OPEN** ステートメントによって、新しいファイルを作成することもできます。

OPEN ステートメントの構文は、次のとおりです。

OPEN *file$* [**FOR** *mode*] [**ACCESS** *access*] [*lock*] **AS** [#] *filenumber%* [**LEN** = *reclen%*]

引数 *file$* で、オープンするファイル名を指定します。ファイル名は、MS-DOS のファイルの名前付け規約に従わなければなりません。ディレクトリ パスやドライブも指定できます。*mode* には、データ入出力のモードを示す 5 つのキーワード (**INPUT, OUTPUT, APPEND, RANDOM, BINARY**) のいずれかを指定します。

次に、**OPEN** ステートメントのいくつかの使い方を示します。

- 新しいデータ ファイルを作成して、データを書き込む。

```
'PRICE.DAT というファイルがカレント ディレクトリにないとき
OPEN "PRICE.DAT" FOR OUTPUT AS #1
```

- すでに存在するデータ ファイルをオープンし、以前の内容をすべて削除して、新しいデータを書き込む。

```
'PRICE.DAT というファイルが、カレント ディレクトリにすでに存在するとき
'新しいデータの書き込みができますが、古いデータはすべて失われます。
OPEN "PRICE.DAT" FOR OUTPUT AS #1
```

- すでに存在するデータ ファイルをオープンして、ファイルの終わりに新しいデータを追加する。

```
OPEN "PRICE.DAT" FOR APPEND AS #1
```

APPEND モードでは、指定された名前のファイルがカレント ディレクトリに存在しない場合、新しいファイルが作成されます。

- すでに存在するデータ ファイルをオープンし、そのファイルからデータを読み出す。

```
OPEN "PRICE.DAT" FOR INPUT AS #1
```

- すでに存在するデータ ファイルをオープンし (その名前を持つファイルが存在しないときは、新しいファイルを作成し)、固定長のレコードを読み書きする。

```
OPEN "PRICE.DAT" FOR RANDOM AS #1
```

- すでに存在するデータ ファイルをオープンし (その名前を持つファイルが存在しないときは、新しいファイルを作成し)、ファイルの中の任意のバイト位置からデータの読み込みや追加を行う。

```
OPEN "PRICE.DAT" FOR BINARY AS #1
```

ファイル番号

OPEN ステートメントは、ファイルに対してデータの入出力のモードを指定するだけでなく、そのファイルに固有な番号を割り当てます。引数 *filenumber* はファイル番号で、1 ～255 の範囲の整数です。**OPEN** ステートメント以降のファイル I/O ステートメントでは、このファイル番号によってファイルを指定します。ファイルがオープンしている間は、その番号を他のファイルに割り当てることはできません。ファイルをクローズすると、ファイル番号が解放され、他のファイルで利用できるようになります。プログラムでは、同時に複数のファイルをオープンできます。同時にオープンできるファイルの数は、MS-DOS のバージョン，利用可能なバッファの数，ファイルに割り当てられているメモリ量などによって異なります。

FREEFILE 関数を使うと、そのとき使われていない最小のファイル番号を知ることができます。この関数が返した番号は、**OPEN** ステートメントでファイルに割り当てることができます。例えば次のような場合、**FREEFILE** は 3 を返します。

```
OPEN "TEST1.DAT" FOR RANDOM AS #1
OPEN "TEST2.DAT" FOR RANDOM AS #2
FileNum = FREEFILE
OPEN "TEST3.DAT" FOR RANDOM AS #FileNum
```

FREEFILE 関数は、ファイルをオープンする、汎用的なプロシージャを作るときに役立ちます。**FREEFILE** 関数を使うと、すでに使われているすべてのファイル番号をプロシージャに引き渡す必要がありません。

ファイル名

OPEN ステートメントでは、ファイル名として、次の文字を任意に組み合わせた文字列式を指定できます。

- 文字 a ～ z および A ～ Z
- 数字 0 ～ 9
- 次の特殊文字

  ( ) @ # $ % ^ & ! - _ ' ~

文字列式には、ドライブ名やディレクトリ名も指定できます。したがって、別のドライブやカレント ディレクトリ以外のディレクトリにあるデータ ファイルを処理することができます。例えば、次の **OPEN** ステートメントはすべて有効です。

```
OPEN "..¥GRADES.QTR" FOR INPUT AS #1

OPEN "A:¥SALARIES¥1992.MAN" FOR INPUT AS #2

Filename$ = "TempFile"
OPEN Filename$ FOR OUTPUT AS #3

BaseName$ = "Invent"
OPEN BaseName$ + ".DAT" FOR OUTPUT AS #4
```

また、ファイル名はオペレーティング システムによっても制限されます。通常、MS-DOS では、ベース名 (ピリオドの左側にある文字列) として最大 8 文字、拡張子 (ピリオドの右側にある文字列) として最大 3 文字の文字列を指定できます。Basic プログラムの中の長いファイル名は、次のように切り捨てられます。

プログラムの中のファイル名	実際のファイル名
Prog@Data@File	PROG@DAT.A@F ファイル名全体が 11 文字を超えたので、最初の 8 文字をベース名として使い、ピリオド (.) を挿入し、次の 3 文字を拡張子として使います。残りの文字はすべて切り捨てられます。
Mail#.Version1	MAIL#.VER ベース名 (Mail#) は 8 文字以下ですが、拡張子 (Version1) が 3 文字を超えるため、4 文字目以降は切り捨てられます。
RELEASE_NoteS.BAK	ランタイム エラー メッセージ「ファイル名が無効です」が表示されます。ファイル名に拡張子 (この場合は .BAK) が含まれているとき、ベース名は 8 文字以下でなければなりません。

MS-DOS では、ファイル名の大文字小文字を区別しないため、ファイル名の中の小文字はすべて大文字に変換されます。したがって、大文字と小文字を混ぜた名前で、ファイルを区別することはできません。例えば、INVESTOR.DAT という名前のファイルがすでにディスク上に存在するとき、次の **OPEN** ステートメントを実行すると、このファイルに上書きし、ファイルに保管されているすべての情報を消してしまいます。

```
OPEN "Investor.DAT" FOR OUTPUT AS #1
```

FileOpener プロシージャ

ここでは、データ ファイルのオープンを行う FileOpener という名前のプロシージャを考えながら、**INPUT**, **OUTPUT**, **APPEND** の 3 種類のシーケンシャル モード, **RANDOM** アクセス モード, **BINARY** アクセス モードの **OPEN** ステートメントの構文について説明します。このプロシージャは、ファイル名, オープンするモードを表す数値, レコード長を表す整数 (ランダム アクセス ファイルの場合) やバッファのサイズ (シーケンシャル アクセス ファイルの場合) を受け取ります。また、ファイル番号は **FREEFILE** 関数を使って取得します。

FileOpener プロシージャは **FUNCTION** プロシージャであり、ファイルがオープンできるとファイル番号を返します。それ以降のステートメントでは、このファイル番号を使います。また、エラー処理に FileErrors プロシージャを使っていますが、このプロシージャについては、第 9 章「エラー ハンドリング」を参照してください。

FileOpener プロシージャは、アプリケーションにそのまま組み込んで利用することもできますし、アプリケーションでオープンするファイルの種類に応じて、必要な部分だけを使うこともできます。例えば、この章の後で紹介するテキスト エディタ アプリケーションでは、ファイルをオープンするときにシーケンシャル アクセスしか使わないので、FileOpener プロシージャから必要な部分だけを抜き出し、新たにプロシージャを作成しています。

```
FUNCTION FileOpener (OFN$, Mode%, RL%) AS INTEGER
    'Mode% パラメータはファイルのアクセス モードを表す 1～5 の整数です。
    '次のような意味を持ちます。
    '1: シーケンシャル出力モード (OUTPUT)
    '2: シーケンシャル入力モード (INPUT)
    '3: シーケンシャル追加モード (APPEND)
    '4: ランダム アクセス モード (RANDOM)
    '5: バイナリ アクセス モード (BINARY)
    '
    FileNum% = FREEFILE                    'ファイル番号を取得します.
    ON LOCAL ERROR GOTO OpenerError 'エラー処理ルーチンを設定します.
    SELECT CASE Mode%    '引数の値に従って, アクセス モードを選択します.
        CASE 1                          'シーケンシャル出力モード
            OPEN OFN$ FOR OUTPUT AS FileNum%
        CASE 2                          'シーケンシャル入力モード
            OPEN OFN$ FOR INPUT AS FileNum% LEN = RL%
        CASE 3                          'シーケンシャル追加モード
            OPEN OFN$ FOR APPEND AS FileNum%
        CASE 4                          'ランダム アクセス モード
            OPEN OFN$ FOR RANDOM AS FileNum% LEN = RL%
        CASE 5                          'バイナリ アクセス モード
            OPEN OFN$ FOR BINARY AS FileNum%
        CASE ELSE
            EXIT FUNCTION
    END SELECT
    FileOpener = FileNum%        '戻り値としてファイル番号を返します.
EXIT FUNCTION
OpenerError:
    Action% = FileErrors(ERR)
    SELECT CASE Action%
        CASE 0
            RESUME
        'CASE ELSE があるので, CASE 1 または CASE 2 を
        '指定する必要はありません.
        CASE ELSE
            FileOpener = 0       'ファイルをオープンできなかった
                                 'ことを呼び出し側に知らせます.
            EXIT FUNCTION
        END SELECT
END FUNCTION
```

OPEN ステートメントの **LEN** キーワードに続いて指定する値は、ランダム アクセス ファイルの場合はレコード長を示し、シーケンシャル アクセス ファイルでは、一度にメモリに読み込むデータのサイズ(ファイル バッファ サイズ)を示します。バッファ サイズを大きくすると、大きなファイルを扱うときにデータの転送速度が上がりますが、より多くのメモリが必要になります。逆に、バッファを小さくすると必要なメモリは少なくて済みますが、転送速度は低下します。ランダム アクセスの場合のデフォルトは128、シーケンシャル アクセスでは512バイトです。バイナリ アクセスでは、転送する単位はデータを受け取る変数のサイズによって決まり、**LEN** で値を指定しても無効です。

データ ファイルのクローズ

データ ファイルをクローズすると、ファイル バッファ (メモリ内の一時的なデータの保管領域) に格納されているすべてのデータがファイルに書き込まれます。また、そのファイルに割り当てられていたファイル番号は解放され、この番号は別の **OPEN** ステートメントで使えるようになります。

ファイルをクローズするには、**CLOSE** ステートメントを使います。例えば、次のステートメントで PRICE.DAT というファイルをオープンしているとします。

```
OPEN "PRICE.DAT" FOR OUTPUT AS #1
```

この場合、`CLOSE #1` を実行すると、PRICE.DAT への出力は終了します。また、次のステートメントで PRICE.DAT をオープンしているときは、`CLOSE #2` でクローズします。

```
OPEN "PRICE.DAT" FOR OUTPUT AS #2
```

CLOSE ステートメントの引数にファイル番号を指定しないと、そのときオープンされているすべてのファイルがクローズされます。

入出力を行うアプリケーションが終了したとき、オープンされているファイルがあれば、自動的にクローズされます。この他にも、**RUN, CLEAR, RESET, SYSTEM** ステートメントや、独立型実行可能ファイルの **CHAIN** ステートメントによっても、ファイルは自動的にクローズされます。

注　意　1 つのファイルを異なるモードで同時にオープンすることは可能です。ただし、シーケンシャル アクセス ファイルを **OUTPUT** モードや **APPEND** モードでオープンするときは、他のモードで同時にオープンできません。このときは、ファイルを一度クローズして、再度別のモードでオープンします。

同じファイルを異なるファイル番号で同時にオープンした場合、ファイル バッファも別々に確保されるので、読み込みと書き込みを同時に行うとファイルの内容に矛盾が生じる可能性があります。十分注意して利用してください。

12.2　シーケンシャル アクセス ファイルの利用

Visual Basic では、テキスト エディタなどのツールを簡単に作成することができます。テキスト ファイルを扱うときは、シーケンシャル アクセスを使います。

これまで説明したように、ファイルへアクセスするには、まず **OPEN** ステートメントを使ってファイルをオープンします。シーケンシャル アクセスでは、一度に 1 つのファイルをオープンし、ファイルの内容を読み込んだり、上書きしたり、新しいテキストを追加したりできます。オープンしたファイルの処理が終わったら、ファイルをクローズして、別の操作を行うときはもう一度オープンします。

例えば、ファイル中のテキストを編集するときは、**INPUT** モードでファイルをオープンしてから、ファイルの内容を読み込んで、複数行が表示できるテキスト ボックス (MultiLine プロパティが True) に表示し、ファイルをクローズします。テキスト ボック

スで必要な編集を行った後、ファイルを **OUTPUT** モードで再びオープンし、元のファイルの内容をテキスト ボックスのテキストで置き換えます。

注　意　シーケンシャル **INPUT** モードでファイルをオープンするときは、既存のファイル名を指定します。指定したファイルが見つからないときは、エラーになります。これに対して、シーケンシャル **INPUT** 以外のモードで、存在しないファイルを指定してオープンしようとすると、新規にファイルが作成されて、そのファイルがオープンされます。

簡易テキスト エディタ

第 8 章「メニューの作成」では、Visual Basic で提供されているメモ帳アプリケーションを参考に、簡単なテキスト エディタを作成しました。メモ帳アプリケーションも、ここで説明するシーケンシャル アクセス ファイルのプログラムを基に作成されています。

図 12.1 のテキスト エディタは、メモ帳アプリケーションの機能の一部を備えています。このエディタの [ファイル] メニューには、新たに 2 つのコマンドが追加されています。ユーザーは、複数行が表示可能なテキスト ボックスに入力したテキストを、ファイルとしてセーブすることができます。ユーザーが最初にテキストをセーブするとき、mnuFileSaveAs_Click から Getname プロシージャが呼び出され、ファイル名の入力を求めます。指定したファイルへのテキストのセーブは、mnuFileSave_Click プロシージャによって行われます。既存のファイルを編集するときは、mnuFileOpen_Click プロシージャから OpenFile プロシージャが呼び出され、シーケンシャル **INPUT** モードでテキスト ファイルをオープンし、**LINE INPUT #** ステートメントを使って 1 行ずつデータを読み込みます。読み込んだ各行は、変数 Temp$ に追加されます。ファイル内のデータを最後まで読み出すと、変数 Temp$ の内容がテキスト ボックスの Text プロパティに代入されます。また、フォームのサイズを変更するときにテキスト ボックスのサイズを変更するプロシージャを使っています。

ファイルの内容を変更するときは、テキスト ボックスのテキストを編集し、新しいテキストで元のファイルを上書きします。MultiLine プロパティが True (-1) のテキスト ボックスで表示可能な文字数は、32,000 バイトより若干少ない量までです。このテキスト エディタでは、ロードできるファイルのサイズを 30,000 バイトに制限しています。

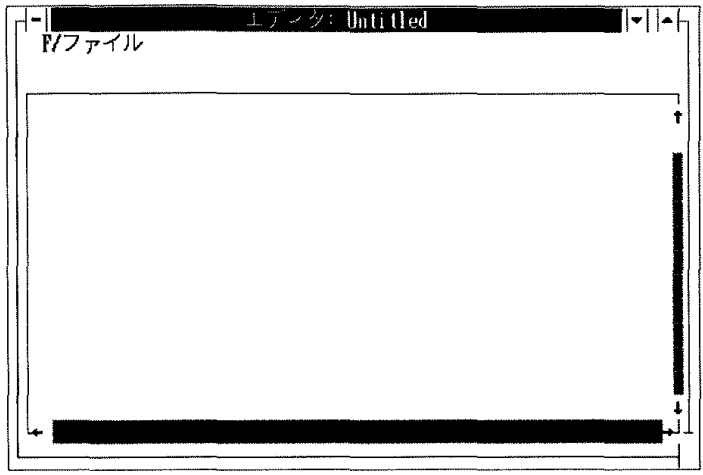


図 12.1 簡単なテキスト エディタ

プロパティの設定

このテキスト エディタ アプリケーションには、ファイルから読み込んだテキストを表示するテキスト ボックス コントロールと、[ファイル] メニューのコマンドを示すメニュー コントロールが含まれます。フォームとテキスト ボックスのプロパティの設定は、次のようになります。

オブジェクト	プロパティの設定
フォーム	FormName = Editor
テキスト ボックス	CtlName = txtEdit Text= (なし) MultiLine = True ScrollBars = 3 (両方)

テキスト ボックスの Multiline プロパティを True (-1) に設定していることに注意してください。これを False (0) に設定すると、テキスト ボックスに複数行のテキストを表示できなくなります。この場合、テキスト ボックスの大きさにかかわらず、テキストを入力した後に [ENTER] キーを押して次の行に移ろうとしても、カーソルはその行から移動しません。また、複数行のテキストを読み込んだとき、行末にキャリッジリターンとラインフィードの ASCII コードに対応する文字 (機種によって異なります) が表示されます。

メニュー コントロールのプロパティの設定は、次の表のようになります。[ファイル] メニューには、テキスト ボックスの現在の内容のクリア，ファイルのオープン，現在のファイルへのテキストの保管，新しいファイルへのテキストの保管，アプリケーションの終了の 5 つのコマンド (メニュー項目) があります。

キャプション (Caption)	コントロール名 (CtlName)	インデント
&F/ファイル	mnuFile	インデントしない
&N/新規	mnuFileNew	レベル 1
&O/オープン	mnuFileOpen	レベル 1
&S/保管	mnuFileSave	レベル 1
&A/新規保管	mnuFileSaveAs	レベル 1
&X/終了	mnuFileExit	レベル 1

テキスト エディタの動作

このテキスト エディタでは、いくつかのイベント プロシージャとジェネラル プロシージャを使います。フォームに対するイベント プロシージャは、アプリケーションの初期化を行います。メニュー コントロールに対するイベント プロシージャは、テキスト ボックス内のテキストのクリア，既存のファイルのロード，テキストのファイルへの保管，アプリケーションの終了を行います。ジェネラル プロシージャは、必要に応じて呼び出されます。

テキスト エディタで使われているプロシージャを次に示します。

プロシージャ	説明
[ファイル] メニューのイベント プロシージャ mnuFileNew_Click mnuFileOpen_Click mnuFileSave_Click mnuFileSaveAs_Click mnuFileExit_Click	[ファイル] メニューのコマンドは、Click イベントに対するイベント プロシージャで処理されます。例えば、ユーザーが [終了] コマンドを選択すると、mnuFileExit_Click イベント プロシージャが呼び出され、テキスト ボックスのテキストが変更されているかどうかをチェックし、変更されているときは変更内容を保管するかどうかの選択を求めます。ユーザーの指示に従ってテキストを保管または破棄したあと、すべてのファイルをクローズし、フォームをアンロードしてアプリケーションを終了します。他のメニュー項目のプロシージャも、その名前が表す操作を行います。
Form_Load	アプリケーションの起動時に初期化を行い、アプリケーションの動作内容とその操作方法をユーザーに示します 。
SaveChanges	FileChanged フラグは、テキスト ボックスのテキストを変更しているかどうかを示しますが、このフラグが真のときは、SaveChanges プロシージャを使ってテキストを保管できます。
GetName	オープンまたは保管するファイルの名前を指定します。またはファイル名をリネームします。
OpenFile	ユーザーが指定したファイルをオープンし、その内容をテキスト ボックスに読み込みます。
txtEdit_Change	テキスト ボックスの内容が変更されたときに、FileChanged フラグを真に設定します。

モジュールレベル コード

テキスト エディタのフォーム モジュールのモジュール レベルでは、次のような宣言を行います。

```
DECLARE SUB mnuFileSave_Click ()
DECLARE FUNCTION SaveChanges () AS INTEGER
DECLARE FUNCTION FileChanges () AS INTEGER
DECLARE SUB mnuFileNew_Click ()
DECLARE SUB GetName (Operation$, OFN$, Cancel%)
DECLARE FUNCTION OpenFile (QualifiedName$) AS INTEGER
DECLARE SUB mnuFileSaveAs_Click ()
'定数を定義します.
CONST FALSE = 0, TRUE = NOT FALSE
'このフォーム モジュール全体に有効な変数を宣言します.
DIM SHARED NLC$
DIM SHARED FileChanged%
DIM SHARED OFN$
```

Form_Load プロシージャ

```
SUB Form_Load ()
    '変数を初期化します.
    OFN$ = "Untitled"
    Caption = "エディタ: " + OFN$
    FileChanged% = FLASE
    NLC$ = CHR$(13) + CHR$(10)
END SUB
```

mnuFileExit_Click プロシージャ

Form_Load イベント プロシージャ以外で、アプリケーションが応答しなければならないイベントは、[ファイル] メニューの各コマンドの Click イベントだけです。もっとも簡単な Click イベント プロシージャは、ユーザーが [ファイル] メニューの [終了] コマンドを選択したときに呼び出される mnuFileExit_Click イベント プロシージャで、このプロシージャによりアプリケーションが終了します。テキストを変更しているときは、必要に応じて SaveChanges を呼び出して、現在のテキストを保管できます。

```
'[終了] コマンドの Click イベント プロシージャ
'アプリケーションを終了します.
SUB mnuFileExit_Click ()
    '変更されているファイルを保管した後, プログラムを終了します.
    IF SaveChanges THEN
        CLOSE           'すべてのファイルをクローズします.
        SCREEN.HIDE
        END             'アプリケーションを終了します.
    END IF
END SUB
```

SaveChanges プロシージャ

SaveChanges プロシージャは、テキスト ボックスの内容を保管するかどうかの選択を求めるダイアログ ボックスを表示します。ユーザーが <はい> を選択すると、mnuFileSave_Click イベント プロシージャを呼び出してテキストをファイルに保管し、真を返します。テキストを破棄する <いいえ> を選択すると、mnuFileSave は呼び出されませんが、SaveChanges は真を返します。また、<取消> を選択すると、SaveChanges は偽を返し、何も実行しないで元に戻ります。

```
FUNCTION SaveChanges () AS INTEGER
    SaveChanges = TRUE
    '最後に保管した後でファイルを変更している時は変更内容を
    '保管するかどうかユーザーに変更を求めます.
    IF FileChanged% = TRUE THEN
        x% = MSGBOX("保管しますか: " + OFN$ + "?", 3, "エディタ")
        '操作を取り消すと, 偽を返します.
        IF x% = 2 THEN
            SaveChanges = FALSE
        '変更を保管する時は, ファイル保管プロシージャを呼び出します.
        ELSEIF x% = 6 THEN
            mnuFileSave_Click
            'ファイルの保管を中止すると, 偽を返します.
            IF OFN$ = "Untitled" THEN SaveChanges = FALSE
        END IF
    END IF
END FUNCTION
```

mnuFileNew_Click プロシージャ

```
SUB mnuFileNew_Click ()
    '必要に応じてテキストの変更内容を保管し, 変数を初期化します.
    IF SaveChanges THEN
        OFN$ = "Untitled"
        Caption = "エディタ: " + OFN$
        txtEdit.Text = ""
        FileChanged% = FALSE
    END IF
END SUB
```

mnuFileOpen_Click プロシージャ

mnuFileOpen_Click プロシージャは、ディスク上の既存のテキスト ファイルをロードします。このプロシージャは、OpenFile, SaveChanges, GetName を呼び出します。OpenFile は、シーケンシャル **INPUT** モードでファイルをオープンします。

```
SUB mnuFileOpen_Click ()
    'オープンするファイル名を取得します. メモ帳アプリケーションでは,
    'コモン ダイアログ ツールキットを利用していますが, ここでは,
    '単純化のために, GetName プロシージャを呼び出しています.

    GetName "オープンするファイル名は?", NewOFN$, Cancel%

    IF NOT Cancel% THEN
        '必要に応じて変更内容を保管してから, 指定されたファイルを
        'オープンします.
```

```
        IF SaveChanges THEN
            OFN$ = NewOFN$
            IF NOT OpenFile(OFN$) THEN
                 'ファイルがオープンできなければ，新しいファイルを作成
                 'します．
                mnuFileNew_Click
            ELSE
                 'オープンしたファイル名でキャプションを更新します．
                Caption = "エディタ: " + OFN$
                FileChanged% = FALSE
                 'まだファイルは更新されていません．
            END IF
        END IF
    END IF
END SUB
```

GetName プロシージャと OpenFile プロシージャ

GetName は、ファイル名の入力を求める単純なプロシージャです。メモ帳アプリケーションでは、コモン ダイアログ ボックス (CMNDLGF.FRM と CMNDLG.BAS) を使ってより複雑な処理を行っています。コモン ダイアログ ボックスを使うと、必要とするファイルがカレント以外のディレクトリやドライブにあるときでも、ファイルを簡単に読み込むことができます。コモン ダイアログ ツールキットの使い方については、第5章「ダイアログ ボックスの作成」を参照してください。

```
SUB GetName (Operation$, OFN$, Cancel%)
    'ファイル名の入力を要求します．
    NameToUse$ = INPUTBOX$(Operation$, "ファイル名", OFN$)
    IF NameToUse$ <> "" THEN
        'ファイル名を大文字に変換します．
        OFN$ = LTRIM$(RTRIM$(UCASE$(NameToUse$)))
        Cancel% = FALSE
    ELSE
        Cancel% = TRUE
    END IF
END SUB
```

ユーザーが操作を取り消した場合を除き、OpenFile プロシージャを呼び出して、ファイルからテキストを読み込みます。

```
FUNCTION OpenFile (QualifiedName$) AS INTEGER
    'エラー処理ルーチンを設定します．
    ON LOCAL ERROR GOTO OpenFileErr
    IF txtEdit.Text <> "" THEN txtEdit.Text = ""
    '新しいファイルをシーケンシャル INPUT モードでオープンします．
    filenum% = FREEFILE
    OPEN QualifiedName$ FOR INPUT AS filenum%
    'ファイルの内容をテキスト ボックスに読み込みます．
    'ファイルが大きすぎて 30K バイトを超えるときはメッセージを表示
    'します．
    IF LOF(filenum%) < 30000 THEN
        Temp$ = ""
        DO UNTIL EOF(filenum%)
            LINE INPUT #filenum%, dummy$
            Temp$ = Temp$ + dummy$ + NLC$
```

```
        LOOP
    ELSE
        MSGBOX "ファイルが大きすぎて，エディタに読み込むことができません"
        OpenFile = FALSE
        EXIT FUNCTION
    END IF
    CLOSE #filenum%
    txtEdit.Text = Temp$
    OpenFile = TRUE
EXIT FUNCTION

OpenFileErr:
    MSGBOX ERROR$ + NLC$ + "オープンできません", 0, "エディタ"
    OpenFile = FALSE
    '強制的にこのプロシージャから抜け出します.
    EXIT FUNCTION
END FUNCTION
```

ファイルをオープンしたあと、**LINE INPUT** ステートメントを使ってファイルの内容をテキスト ボックスに読み込みます。複数行が表示可能な (MultiLine プロパティが -1 の) テキスト ボックスで保持できるデータは、最高 32,000 バイトであることに注意してください。このアプリケーションでは、ファイルのサイズが 30,000 バイトを超えるときは、テキスト ボックスに表示しきれないことを示すメッセージを表示しています。

mnuFileSave_Click プロシージャ

[ファイル] メニューの [保管] コマンドは、そのときタイトル バーに名前が表示されているファイルにテキストを保管します。エラー処理を有効にしてから、ファイルをシーケンシャル **OUTPUT** モードでオープンし、**PRINT #** ステートメントを使って、テキスト ボックス (txtEdit) の内容でカレント ファイルを上書きします。シーケンシャル **APPEND** モードでファイルをオープンすると、テキストボックスの内容がファイルに追加されることに注意してください。

```
SUB mnuFileSave_Click ()
    'エラー処理ルーチンを設定します.
    ON LOCAL ERROR GOTO SaveFileErr
    'ファイル名が設定されていなければ，新しいファイル名の入力を要求します.
    IF OFN$ = "Untitled" THEN
        mnuFileSaveAs_Click
    ELSE
        'シーケンシャル OUTPUT モードでファイルをオープンします.
        filenum% = FREEFILE
        OPEN OFN$ FOR OUTPUT AS filenum%
        'テキスト ボックスの内容をファイルに出力します.
        PRINT #filenum%, txtEdit.Text
        CLOSE #filenum%
        FileChanged% = FALSE
    END IF
EXIT SUB
```

```
SaveFileErr:
    MSGBOX ERROR$ + NLC$ + "保管できません", 0, "エディタ"
    '強制的にこのプロシージャから抜け出します.
    EXIT SUB
END SUB
```

mnuFileSaveAs_Click プロシージャ

元のファイルを残して、新しい名前でファイルを保管したい場合は、[新規保管] コマンドを使います。mnuFileSaveAs_Click プロシージャは、新しいファイル名の入力を求めたあと、mnuFileSave_Click プロシージャを呼び出してファイルを保管します。このとき、フォームのタイトル バーには新しい名前が表示されます。また、txtEdit 内のテキストはディスク ファイルの内容と一致するため、FileChanged フラグは偽に設定されます。

```
SUB mnuFileSaveAs_Click ()
    GetName "新しいファイル名を入力してください.", NewOFN$, Cancel%
    IF NOT Cancle% THEN
        OFN$ = NewOFN$
        Caption = "エディタ: " + OFN$
        mnuFileSave_Click
        FileChanged% = FALSE
    END IF
END SUB
```

ファイルを新しい名前で保管するときは、元のファイルは上書きされずにそのまま残ります。元のファイルを削除するときは、プログラム内で **KILL** ステートメントを使います。

コモン ダイアログ ツールキットの使い方

GetName プロシージャの代わりにコモン ダイアログ ツールキットを使うこともできます (コモン ダイアログ ツールキットについては、第 5 章「ダイアログ ボックスの作成」を参照してください)。だたし、ファイル名およびパスを示す文字列変数 FileName と PathName は、グローバルに定義 (メモ帳アプリケーションのように **DIM SHARED** と宣言) されていなければなりません。メモ帳アプリケーションでは、数カ所でコモン ダイアログ ツールキットを利用していますから、参考にしてください。

テキスト ボックスでの [TAB] キーの効果

フォームを使ったアプリケーションでは、別のコントロールにフォーカスを移すには [TAB] キーを使います。複数行が表示可能なテキスト ボックス内で [TAB] キーを押すと、デフォルトでは複数のスペースは挿入されず、次のコントロールにフォーカスが移ってしまいます。複数行が表示可能なテキスト ボックスで、[TAB] キーを使ってスペースを入力できるようにするには、テキスト ボックスがフォーカスを持つとき、他のすべてのコントロールの TabStop プロパティを False に設定します。ただしこの場合、ユーザーの操作でフォーカスを移すことができなくなる点に注意してください。他のコントロールの TabStop プロパティは True のままにして、テキスト ボックス内では [TAB] キーの代わりに [CTRL]+[I] キーを押して複数のスペースを入力することもできます。

12.3 ランダム アクセス ファイルの利用

ランダム アクセス ファイルは、住所録や仕入台帳のように、大量の定形のデータを扱うときに便利です。1つ1つのデータ項目はフィールドに格納され、フィールドが複数集まってレコードを構成します。さらにレコードが複数集まって、ランダム アクセス ファイルを構成します。

ランダム アクセス ファイルのレコード

シーケンシャル アクセス ファイルでは、データの区切りとして改行やカンマが使われるのに対し、ランダム アクセス ファイルでは、このようなデータの区切りは必要ありません。ランダム アクセス ファイルのレコードと、レコード中のフィールドは、それぞれ固定長で定義されます。これによって、例えば3番目のレコードがファイルの先頭から何バイト目から始まるかを計算で求めることができ、すべてのレコードにすばやくアクセスすることができます。次の図は、ランダム アクセス ファイルのレコードとフィールドを示しています。

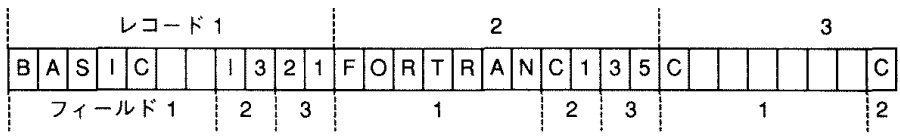


図 12.2　ランダム アクセス ファイルのレコード

ランダム アクセス ファイルのデータの作成と追加

ランダム アクセス ファイルを作成してデータを記録するには、次のようにします。

1. 各レコードのフィールドを定義します。
2. ファイルを **RANDOM** モードでオープンし、各レコードのレコード長を指定します。
3. 新しいレコードにデータを設定し、そのレコードをファイルに書き込みます。

レコードとフィールドの定義

レコードは、**TYPE...END TYPE** ステートメントを使って定義する、ユーザー定義型変数を使って表します。ユーザー定義型変数は、レコード変数や構造体と呼ばれることもあります。ユーザー定義型は、整数や倍精度浮動小数点数などの標準のデータ型の集まりである、一種の複合データ型です。ランダム アクセス ファイルのデータを読み書きするとき、1つのレコードが1つのユーザー定義型に対応し、レコード中の各フィールドがユーザー定義型変数の各要素に対応します。ユーザー定義型の変数を利用するには、まずユーザー定義型を定義しなければなりません。変数の宣言は、プログラムのどの場所でも可能です。ユーザー定義型は、例えば次のように定義します。

```
'レコードを保持するデータ型を定義します.
TYPE TransactionType
    RecordNum        AS LONG              'ファイル中のレコードの位置を
                                          '保持します.
    AccountNo        AS LONG              '口座番号識別子
    AccountID        AS STRING * 80       '口座の名義
    Identifier       AS STRING * 50       '支払い先または預け入れ元
    Comment          AS STRING * 100      '口座名義人に関するコメント
    Amount           AS CURRENCY          '取引金額
    TransactionID    AS INTEGER           '小切手番号または預金 ID
                                          '番号
    CreditOrDebit    AS INTEGER           '支払いのときは -1, 預金の
                                          'ときは +1
    Month            AS INTEGER
    Day              AS INTEGER
    Year             AS INTEGER
END TYPE
```

注 意 ユーザー定義型の中の文字列は、固定長文字列でなければなりません。

変数のサイズ

ランダム アクセス ファイルを設計するとき、各フィールドに最大どれくらいの長さのデータが格納される可能性があるかを十分考慮する必要があります。数値データであれば、格納する数値の最大値と最小値，精度を考えて、適切なデータ型を選んでください。例えば、年月日は整数型の範囲を超えることはありませんが、データの件数 (レコード数) を表す値は、32,767 個を超える可能性があります。このため、この例では RecordNum を長整数型として定義しています。また、取引金額は計算誤差がでないように固定小数点である通貨型を使っています。

人名や会社名などのデータは、それぞれ長さが異なります。読み込んだ文字列データがフィールドよりも長いと、余分な文字列は切り捨てられます。そのため、文字列を入力するフィールドは、あらかじめ適切な余裕を持った長さで定義します。文字列データが定義されているフィールドの長さよりも短いときは、フィールドの残りの部分は自動的にスペース (**CHR$**(32)) で埋められます。

数値を含むレコードを格納するときは、ランダム アクセス ファイルを使うと、シーケンシャル アクセス ファイルを使ったときよりもディスク スペースを節約できます。これは、シーケンシャル アクセス ファイルが数値を文字列として保管するのに対し、ランダム アクセス ファイルでは数値そのものとして保管するからです。

例えば、17000 という数値は、シーケンシャル アクセス ファイルの中では 1 桁を 1 バイトとして 5 バイトで表されます。一方、ランダム アクセス ファイルでは、整数であれば 2 バイト，長整数と単精度実数は 4 バイト，倍精度実数と通貨型は 8 バイトを使って表します。17000 が整数型なら、ランダム アクセス ファイルでは 3 バイト節約できることになります。

ユーザー定義型を定義するときは、固定長文字列の長さを調整して、ユーザー定義型の全体のサイズが 2 のべき乗になるようにします。例えば、大量のデータのソートを行いたいとき、すべてのデータを配列に格納できれば便利です。配列変数のサイズは、1 つの要素が 2 のべき乗であれば、より大きく確保することができます。

ユーザー定義型変数の宣言

ユーザー定義型を定義したら、そのデータ型を持つ変数を宣言します。

```
'TransactionType 型の変数 RecordVar を宣言します.
DIM SHARED RecordVar AS TransactionType
```

さまざまなプロシージャでファイル中のデータを扱うときは、**SHARED** キーワードを指定してモジュール全体で有効な変数として宣言します。また、複数のモジュールで共有したいときは、**COMMON SHARED** で宣言します。

ファイルのオープンとレコード長の指定

ランダム アクセス ファイルのレコードは固定長なので、各レコードの長さをプログラムの中で指定しなければなりません。指定しないと、レコード長はデフォルトの 128 バイトになります。レコード長は、**OPEN** ステートメントの中で、**LEN** キーワードの後にバイト単位で指定します。ランダム アクセス ファイルをオープンするときの構文を次に示します。

OPEN *file$* **FOR RANDOM AS** [#] *filenumber%* **LEN** = *Reclen%*

「データ ファイルのオープン」で説明した FileOpener プロシージャでは、ランダム アクセス ファイルをオープンする **OPEN** ステートメントで **LEN** キーワードを指定していました。

ランダム アクセス ファイルのメモリ バッファは、ファイルのレコード サイズと、ファイルからデータを受け取るときに使うユーザー定義型変数のサイズに一致していなければなりません。**LEN** 関数を使うと、ユーザー定義型変数のサイズを調べることができます。

```
OPEN REC.DAT FOR RANDOM AS #1 LEN=LEN(RecordVar)
```

ランダム アクセス ファイルでの GET および PUT ステートメントの使い方

ランダム アクセス ファイルからデータを読み出すには、**GET** ステートメントを使います。また、データを書き込むには、**PUT** ステートメントを使います。これらのステートメントの構文を次に示します。

GET [#] *filenumber%*, [*recordnumber&*], *recordvariable*

PUT [#] *filenumber%*, [*recordnumber&*], *recordvariable*

recordnumber& 引数は、読み込みまたは書き込みを行うレコードのレコード番号です。また、*recordvariable* 引数には、アクセスするランダム アクセス ファイルのレコードの構造に一致したユーザー定義型の変数を指定します。ランダム アクセスでは、データのファイル中の位置をレコード番号で指定して読み書きを行うことができます。**GET** ステートメントで引数 *recordnumber&* の指定を省略すると、次のレコードの各フィールドが、ユーザー定義型変数 *recordvariable* に読み込まれます。また、**PUT** ステートメントで引数 *recordnumber&* の指定を省略すると、*recordvariable* の内容が次のレコードに書き込まれます。

12.4 バイナリ ファイルの I/O

ファイルのデータを読み書きする方法として、ランダム アクセスやシーケンシャル アクセス以外に、バイナリ アクセス (バイナリ ファイルの I/O) があります。バイナリ アクセス モードでファイルをオープンするには、次のように指定します。

OPEN *file$* **FOR BINARY AS #** *filenumber%*

バイナリ アクセスを使うと、テキスト ファイルだけでなく、あらゆるファイルからバイト単位でデータを読み込むことができます。このため、アプリケーション固有のフォーマットで記録されているデータ ファイルや実行可能 (.EXE) ファイルなど、テキスト以外の形式で保管されたファイルの読み込みや変更を行うことができます。

バイナリ アクセス用にオープンしたファイルは、書式化されていないバイトの並びとして扱われます。ファイル中の各バイトを個別のレコードと考えない限り、バイナリ I/O でレコードを扱うことはありません。

バイナリ アクセスとランダム アクセスの違い

バイナリ アクセスでは、ランダム アクセスと同じように、**OPEN** ステートメントを 1 度実行するだけで、ファイルの読み込みと書き込みの両方を行うことができます。シーケンシャル アクセスでは、読み込みと書き込みを切り換えるとき、いったんファイルをクローズしてから、再び同じファイルをオープンしなければなりません。またバイナリ アクセスは、ランダム アクセスと同じように、オープンしたファイル中の操作対象の位置を自由に移動できます。バイナリ アクセスでは、ランダム アクセス ファイルの読み書きで使うステートメントと同じステートメントを使うことができます。

GET # *filenumber%* , [*position&*], *variable*

PUT # *filenumber%* , [*position&*], *variable*

引数 *variable* には、可変長文字列やユーザー定義型を含む任意のデータ型を指定できます。*position&* には、次の **GET** または **PUT** ステートメントで操作を行うファイルの中の位置を、ファイルの先頭バイトを 1 としたバイト数で指定します (ファイル中の第 1 バイトは 1 , 第 2 バイトは 2 になります)。**GET** または **PUT** ステートメントを連続して実行するときに *position&* の指定を省略すると、ファイル ポインタは先頭バイトから最後のバイトまで順番に移動します。

GET ステートメントは、引数 *variable* と同じ長さのバイトをファイルから読み込みます。同様に **PUT** ステートメントは、*variable* と同じバイト数をファイルに書き込みます。例えば、*variable* が整数型であれば、**GET** は 2 バイトを読み込んで *variable* に格納します。また、*variable* が単精度実数型であれば、**GET** は 4 バイトを読み込みます。このとき、**GET** または **PUT** ステートメントの中で引数 *position&* の指定を省略すると、*variable* の長さと同じ距離だけファイル ポインタが移動します。

BINARY モードでオープンしたファイルからデータを読み出すには、**GET** ステートメントか **INPUT$** 関数しか使うことができません。データを書き込むには、**PUT** ステートメントしか使うことができません。

バイナリ アクセス モードでは、ランダム アクセス モードと異なり、ファイルの中の任意のバイト位置に移動し、必要なバイト数だけ読み書きを行うことができます。これに対しランダム アクセスは、レコードの境界に移動し、一定のバイト (レコード長) 単位で読み書きを行うことしかできません。

次に示す BinaryCopy 関数は、バイナリ アクセスを使ってファイルのコピーを行います。この関数は、コピー元のファイルからデータを読み込んでバッファに格納し、そのバッファの内容を指定するファイルに書き込みます。したがって、この関数を使うと、あらゆる種類のファイルのコピーを実行できます。バッファは、30,000 バイトの文字列変数なので、**GET** ステートメントでファイルから読み出されるバイトがバッファに入り切らないことはほとんどありません。ファイルが 30,000 バイトよりも大きいときは、複数回に分けて読み込みと書き込みを行います。この例では、**GET** または **PUT** ステートメントの引数 *position* が省略されているので、ファイル ポインタの位置は順番に移動します。

```
FUNCTION BinaryCopy (SourceName$, TargetName$) AS INTEGER
    'この関数は，ファイルをバイナリ アクセス モードでオープンし，ファイルの
    'サイズを調べ，その内容を文字列変数に割り当てる (必要に応じて複数の部分
    'に分割する) ことによって、すべての種類のファイルをコピーします．
    'コピーが終了すると，真 (-1) を返します．
    '
    'エラー処理ルーチンを設定します．
    ON LOCAL ERROR GOTO BinaryCopyError
    CONST binaryFile = 5
    DIM BufferVar AS STRING
    'コピー元とコピー先のファイルをオープンします。
    SourceNum% = FileOpener(SourceName$, binaryFile, 0)
    TargetNum% = FileOpener(TargetName$, binaryFile, 0)
    'ファイル中のバイトを格納する変数を作成し，スペースで埋めます．
    BufferVar = STRING$(30000, " ")
    '複数の部分に分割する必要があるときは，その個数と最後の部分のサイズを
    '求めます．
    BigChunks% = LOF(SourceNum%) ¥ LEN(BufferVar)
    LeftOver% = LOF(SourceNum%) MOD LEN(BufferVar)

    'ソース ファイルから部分ごとにデータを読み込み，ファイルに書き込みます．
    FOR ind% = 1 TO BigChunks%
        GET SourceNum%, , BufferVar
        PUT TargetNum%, , BufferVar
    NEXT ind%
    'ファイルがバッファ サイズで割り切れないとき，最後の部分を読み込んで
    'ファイルに書き込みます．
    IF LeftOver% THEN
        BufferVar = ""
        BufferVar = STRING$(LeftOver%, " ")
        GET SourceNum%, , BufferVar
        PUT TargetNum%, , BufferVar
    END IF
```

```
IF LOF (SourceNum%) = LOF(TargetNum%) THEN BinaryCopy = -1
    CLOSE
EXIT FUNCTION
BinaryCopyError:
    'エラー処理ルーチン
      .
      .
      .
END FUNCTION
```

SEEK によるファイル ポインタの移動

Visual Basic では、**SEEK** ステートメントと **SEEK** 関数が用意されています。**SEEK** ステートメントは、実際の入出力を行わずに、ファイルの中の任意の位置にファイル ポインタを移動したいときに使います。**SEEK** ステートメントの構文を次に示します。

SEEK [#] *filenumber%, position&*

SEEK ステートメントを実行すると、ファイル番号 *filenumber%* でオープンしたファイルの次の読み込みまたは書き込み操作は、*position&* で指定したバイトから行われます。

SEEK ステートメントに対応するものとして、**SEEK** 関数があります。

SEEK (*filenumber%*)

SEEK 関数は、次に読み込みまたは書き込み操作を行うバイト位置を返します (バイナリ I/O を使ってファイルにアクセスするとき、**LOC** 関数と **SEEK** 関数は同じような結果を返します。**LOC** 関数が、最後に読み書きを行ったバイト位置を返すのに対して、**SEEK** 関数は、次に読み書きを行うバイト位置を返します)。

QLBVIEW アプリケーションは、Quick ライブラリをオープンし、ライブラリに含まれているプロシージャの名前を調べて表示します。次の例 (モジュール レベル コードと GetData プロシージャ) は、バイナリ モードのファイルにアクセスする方法を示しています。これ以外のプロシージャでは、フォームの初期化，ファイル名の入力，データのプリンタへの出力などを行っています。バイナリ ファイルをオープンするとき、レコード長を指定する引数は使わないことに注意してください。**OPEN** ステートメントでレコード長を指定しても無視されます。

```
'QLBVIEW アプリケーションのモジュールレベル コード
'ファイルの情報を保持するためのユーザー定義型を定義します.
TYPE ExeHdr                          'MS-DOS の .EXE ファイルのヘッダ
    other1      AS STRING * 8        '他のヘッダ情報
    CParHdr     AS INTEGER           'ヘッダのサイズ (パラグラフ単位)
    other2      AS STRING * 10       '他のヘッダ情報
    IP          AS INTEGER           'IP の初期値
    CS          AS INTEGER           'CS の初期値 (相対値)
END TYPE

TYPE QBHdr                           '.QLB ファイルのヘッダ
    QBHead      AS STRING * 6        'QB 特有のヘッダ
    Magic       AS INTEGER           'マジック ワード:Quick ライブラリ
                                     'ファイルを識別します.
    SymStart    AS INTEGER           'ヘッダから最初のコード シンボルまでの
                                     'オフセット
```

```
    DatStart     AS INTEGER        'ヘッダから最初のデータ シンボルまでの
                                   'オフセット
END TYPE

TYPE QbSym                         'QuickLib シンボルのエントリ
    Flags        AS INTEGER        'シンボル フラグ
    NameStart    AS INTEGER        '名前テーブルのオフセット
    other        AS STRING * 4     '他のヘッダ情報
END TYPE

'これまでに定義したユーザー定義型の変数を宣言します.
DIM SHARED EHdr AS ExeHdr, QHdr AS QBHdr, QHdrPos AS LONG
DIM SHARED FileName$, PathName$
DIM SHARED FileNum%


SUB cmdOpen_Click ()
'コモン ダイアログ ボックスの FileOpen を使って, ファイル名の入力を求め
'ます.
    FileOpen FileName$, PathName$, "*.QLB", "", (ForeColor),
 ⇒ (BackColor), 0, Cancel%
'ユーザーが取り消しを行わなければ、Quick ライブラリの内容を表示します。
    IF NOT Cancel% THEN
        '"¥" は完全なファイル名を作成するために使います。
        IF RIGHT$(PathName$, 1) <> "¥" THEN Connector$ = "¥"
        lblName.Caption = '名前:' + FileName$
        lblPath.Caption = 'パス:' + PathName$
        GetData PathName$ + Connector$ + FileName$
END SUB

SUB GetData (QuickLib AS STRING)
    'リスト ボックスの内容をクリアします。
    FOR I% = 0 TO lstProcedures.ListCount - 1
        lstProcedures.REMOVEITEM 0
    NEXT I%

    'Quick ライブラリをオープンします。
    FileNum% = FREEFILE
    OPEN QuickLib FOR BINARY AS #FileNum%

    'EXE フォーマット ヘッダを読み込みます。
    GET #FileNum%, , EHdr
    QHdrPos = (EHdr.CParHdr + EHdr.CS) * 16 + EHdr.IP + 1

    'Quick ライブラリ フォーマット ヘッダを読み込みます.
    GET #FileNum%, QHdrPos, QHdr
    IF QHdr.Magic <> &H6C75 THEN
        MSGBOX QuickLib + ": Quick ライブラリではありません"
    ELSE
        DIM QlbSym AS QbSym
        DIM NextSym AS LONG, CurrentSym AS LONG

        '最初のシンボル エントリの位置を計算し, その情報を読み込みます.
        NextSym = QHdrPos + QHdr.SymStart
        GET #FileNum%, NextSym, QlbSym
```

```
        DO
            '次のシンボルの位置を保存します.
            NextSym = SEEK(FileNum%)
            CurrentSym = QHdrPos + QlbSym.NameStart
            'SEEK を使って, 現在のシンボル エントリの名前に移動します.
            SEEK #FileNum%, CurrentSym
            'もっとも長い正規の文字列と、最後の null 文字 (CHR$(0))
            'に対応する 1 バイトを読み込みます.
            Prospect$ = INPUT$(40, #FileNum%)
            'null 文字で終わる名前を取り出します.
            SName$ = MID$(Prospect$, 1, INSTR(Prospect$,
             ⇒  CHR$(0)))
            '先頭に "__", "$", "b$" が付かない名前だけを出力します.
            'これらの文字が先頭に付く名前は通常, 予約語とみなされます.
            IF MID$(temp$, 1, 1) <> "_" AND MID$(temp$, 1, 1)
             ⇒ <> "$" AND UCASE$(MID$(temp$, 1, 2)) <> "B$"
             ⇒ AND LEFT$(temp$, 8) <> "FO_TEXT_" THEN
                IF MID$(temp$, 1, 12) <> "GETCONTAINER" AND
                 ⇒ MID$(temp$, 1, 11) <> "GETPROPERTY" AND
                 ⇒ MID$(temp$, 1, 11) <> "INVOKEEVENT" AND
                 ⇒ MID$(temp$, 1, 12) <> "INVOKEMETHOD" THEN
                    IF MID$(temp$, 1, 11) <> "SETPROPERTY" AND
                     ⇒ MID$(temp$, 1, 9) <> "SETUEVENT" AND
                     ⇒ MID$(temp$, 1, 12) <> "STRINGASSIGN"
                     ⇒ AND MID$(temp$, 1, 13) <>
                     ⇒ "STRINGRELEASE" THEN
                        IF MID$(temp$, 1, 13) <>
                         ⇒ "STRINGADDRESS" AND MID$(temp$, 1,
                         ⇒ 12) <> "STRINGLENGTH" THEN
                            lstProcedures.ADDITEM SName$
                        END IF
                    END IF
                END IF
            END IF
            IF MID$(SName$, 1, 1) <> "_"
                IF MID$(SName$, 1, 1) <> "$"
                    IF UCASE$(MID$(SName$, 1, 2)) <> "B$" THEN
                        lstProcedures.ADDITEM SName$
                    END IF
                END IF
            END IF
            'シンボル エントリを読み込みます.
            GET #FileNum%, NextSym, QlbSym
        'Flags=0 (false) はテーブルの最後を示します.
        LOOP WHILE QlbSym.Flags
    lstProcedures.SETFOCUS
    END IF
    CLOSE #FileNum%
END SUB
```

シーケンシャル アクセスとランダム アクセスでの SEEK の使い方

SEEK ステートメントと **SEEK** 関数は、シーケンシャル アクセスまたはランダム アクセスのためにオープンしたファイルに対しても実行できます。シーケンシャル アクセスでは、**SEEK** ステートメントと **SEEK** 関数はバイナリ アクセスと同じように動作します。**SEEK** ステートメントは、指定したバイト位置にファイル ポインタを移動し、**SEEK** 関数は、次に読み書きを行うバイト位置を返します。

ランダム アクセスのためにファイルをオープンすると、**SEEK** ステートメントは任意のレコードの先頭にファイル ポインタを移動しますが、レコードの中の任意のバイトには移動できません。また、ランダム アクセス ファイルを処理するとき、**SEEK** 関数は、次のバイト位置ではなく次のレコード番号を返します。

12.5　デバイスの操作

Visual Basic は、デバイスの入出力 (I/O) をサポートしています。周辺装置は、ディスク上のデータ ファイルと同じように、入出力のためにオープンできます。

Visual Basic がサポートするデバイスを表 12.1 に示します。CONS:, SCRN:, KYBD: デバイスは、フォームが表示されていないときにだけアクセスできることに注意してください。フォームが表示されているときは、デスクトップの設定を行う **SCREEN** オブジェクト，KeyPress, KeyDown, KeyUp イベントによるキーボード入力，Click, DblClick, MouseDown, MouseUp, MouseMove イベントによるマウス操作などを利用します。また、**PRINTER** オブジェクトを使うと、プリンタに関する多くの命令を簡単に指定できます。

表 12.1　Visual Basic の I/O でサポートされているデバイス

名前	デバイス	サポートする I/O モード
COM1:	シリアル ポート 1	入力および出力
COM2:	シリアル ポート 2	入力および出力
CONS:	画面	出力のみ (フォームが非表示のときのみ)
KYBD:	キーボード	入力のみ (フォームが非表示のときのみ)
LPT1:	プリンタ 1	出力のみ
LPT2:	プリンタ 2	出力のみ
LPT3:	プリンタ 3	出力のみ
SCRN:	画面	出力のみ (フォームが非表示のときのみ)

デバイス I/O とファイル I/O の違い

ファイル I/O で使う関数やステートメントは、デバイス I/O に対して異なる動作をします。また、デバイス I/O では使えない関数やステートメントもあります。ファイル I/O とデバイス I/O の違いを次に示します。

- CONS:, SCRN:, LPT*n*: の各デバイスは、シーケンシャル **INPUT** モード，シーケンシャル **APPEND** モード，**RANDOM** アクセス モードではオープンできません。
- KYBD: デバイスは、シーケンシャル **OUTPUT** モード，シーケンシャル **APPEND** モード，**RANDOM** アクセス モードではオープンできません。
- **EOF, LOC, LOF** の各関数は、CONS:, KYBD:, LPT*n*:, SCRN: の各デバイスに対しては使うことができません。
- **EOF, LOC, LOF** の各関数は、COM*n*: シリアル デバイスに対して使うことができますが、これらの関数の戻り値は、データ ファイルの場合とは意味が異なります (詳しくは「シリアル ポートを利用した通信」を参照してください)。

次のフォームを使わないプログラムでは、データ ファイルの場合と同じ構文を使って、シーケンシャル **OUTPUT** モードで LPT1: や SCRN: デバイスをオープンしています。このプログラムは、指定されたファイルからすべての行を読み込み、画面またはプリンタのどちらか選択されたデバイスに出力します。

```
CLS
'出力するファイルの名前を入力します.
INPUT "出力するファイル名を入力してください: ", FileNam$

'ファイル名が入力されないと，アプリケーションは終了します.
'ファイル名が入力されると，そのファイルをシーケンシャル INPUT モードで
'オープンします.
IF FileNam$ = "" THEN END ELSE OPEN FileNam$ FOR INPUT AS #1

'ファイルの出力先 (画面またはプリンタ) を選択します.
'"S" または "P" キーを押されるまで，入力を求めるプロンプトが表示されます.
DO
    '画面上の 2 行 1 桁の位置にプロンプトを表示します.
    LOCATE 2, 1, 1
    PRINT "画面 (S) またはプリンタ (P) のどちらに出力しますか:";

    'プロンプトの後にスペースを 2 つ出力し，表示されている文字を上書き
    'します.
    PRINT SPACE$(2);

    'プロンプトの直後にカーソルを移動して表示します.
    LOCATE 2, 53, 1
    Choice$ = UCASE$(INPUT$(1))          '入力を受け取ります.
    PRINT Choice$
LOOP WHILE Choice$ <> "S" AND Choice$ <> "P"
```

```
'ユーザーが押すキーに従って，プリンタまたは画面をシーケンシャル OUTPUT
'モードでオープンします．
SELECT CASE Choice$
    CASE "P"
        OPEN "LPT1:" FOR OUTPUT AS #2
        PRINT "プリンタにファイルを出力します．"
    CASE "S"
        CLS
        OPEN "SCRN:" FOR OUTPUT AS #2
END SELECT

'選択された出力デバイスの幅を 80 列に設定します．
WIDTH #2, 80

'ファイルのすべての行を出力するまで，ファイルから 1 行ずつ読み込んで，選択
'されたデバイスに出力します．
DO UNTIL EOF(1)
    LINE INPUT #1, LineBuffer$
    PRINT #2, LineBuffer$
LOOP

CLOSE           'ファイルからの読み込みとデバイスへの出力を終了します．
END
```

次に示す QLBVIEW アプリケーションのプロシージャは、上の例とは異なり、**PRINTER** オブジェクトを使ってプリンタにデータを出力しています。

```
SUB cmdPrint_Click ()
    'プリンタをセットアップし，出力オプションを受け取ります．
    FilePrint Copies%, (ForeColor), (BackColor), Cancel%

'ユーザーが取り消しを行わなければ、Quick ライブラリの内容を表示します。
    IF NOT Cancel% THEN
        FOR I% = 1 TO Copies%
            PRINTER.PRINT FileName$ + "に含まれているルーチン:"
            FOR J% = 0 TO lstProcedures.ListCount - 1
                PRINTER.PRINT lstProcedures.List(J%)
            NEXT J%
            PRINTER.NEWPAGE
        NEXT I%
        PRINTER.ENDDOC
    END IF
    lstProcedures.SETFOCUS
END SUB
```

シリアル ポートを利用した通信

OPEN "COM*n*:" ステートメントを使うと、シリアル ポートをオープンして、モデムやシリアル プリンタなどの周辺装置、または他のコンピュータと、シリアル (ビット単位の) 通信を行うことができます。*n* には、1 または 2 (シリアル ポートを 2 つ持つ場合のみ) を指定します。このとき、次のパラメータを指定できます。

- bps (ビット/秒) で表されるデータ転送速度
- 通信エラーの検出 (パリティ チェック) と検出方法

- ストップ ビットの数 (1, 1.5, 2 のいずれか)
- 実際にデータを送信または受信するときのデータ ビット数

通信のためにシリアル ポートをオープンすると、その I/O デバイスから読み込んだデータを格納するために入力バッファが確保されます。これは、速い通信速度で通信するとき、文字データがそれらを処理するスピードよりも速く受信されるためです。入力バッファのデフォルトのサイズは 512 バイトですが、**OPEN "COM*n*:"** ステートメントの **LEN** キーワードを使って変更できます。次の表に示すように、シリアル ポートに対して **EOF, LOC, LOF** の各関数を使うと、入力バッファのサイズに関する情報が得られます。

関数	戻り値
EOF	入力バッファにまだ読み込まれていない文字データがあるかどうか
LOC	入力バッファの中のまだ読み込まれていない文字数
LOF	出力バッファに残っている空き容量 (バイト単位)

通信プログラムでは、扱うすべての文字が意味のあるデータである可能性があるため、シリアル ポートからデータを入力するときに **INPUT #** や **LINE INPUT #** ステートメントを使うと、うまくいかないことがあります。**INPUT #** では、カンマやキャリッジ リターン (場合によっては、スペースやダブル クォーテーション)が見つかると、変数へのデータの読み込みを中止します。同様に **LINE INPUT #** では、キャリッジ リターンが見つかると、読み込みを中止します。これに対し、**INPUT$** 関数はすべての文字を読み込めるため、シリアル ポートから入力を行う場合にはもっとも適しています。

次の例では、まず、ファイル番号 #1 でオープンしたシリアル ポートに対し **LOC** 関数を使い、入力バッファの中のまだ読み込まれていない文字数を調べます。次に、**INPUT$** 関数を使ってこれらの文字を読み込み、文字列変数 ModemInput$ に格納します。

```
ModemInput$ = INPUT$(LOC(1), #1)
```

端末エミュレータ アプリケーション

次のフォームを使わないアプリケーションでは、コンピュータを通信の端末として扱います。コンピュータをモデムと接続し、単純な I/O デバイスとして利用します。このプログラムは、**OPEN "COM1:"** ステートメントと、それに関連するデバイス I/O 関数を使って、次の操作を行います。

- キーボードから入力された文字をモデムに送る
- モデムから返された文字を画面に表示する

キーボードから入力し、画面に表示されている文字は、オープンしている通信チャンネルを使って一度モデムに送られ、再びコンピュータに返されたものです。モデムに受信検出 (RD) と送信検出 (SD) のインジケータが付いているときは、文字を入力するとインジケータが順番に点灯するので、それを確認できます。

番号をダイヤルするときは、キーボードから ATDT (Attention Dial Touch Tone) コマンドまたは ATDP (Attention Dial Pulse) コマンドを入力します (AT コマンドが使えるモデムの場合)。

これ以外のコマンドをモデムに送るときも、キーボードから入力します。

```
DEFINT A-Z

COLOR 15            '画面の色を設定します.
CLS

' [ESC] を押したときに, INKEY$から返される値を定義します.
Quit$ = CHR$(&H1B)
'プロンプトを出して, カーソルを表示します.
OPEN "CONS:" FOR OUTPUT AS 2
LOCATE 1, 1, 1, 14, 15
PRINT #2, CHR$(27) + "[2J";
PRINT #2, TAB(30); " [ESC] で終了します"
PRINT #2, STRING$(80, "_")

'通信ポートをオープンします (2400 ボー, パリティなし, データ長 8 ビット,
'ストップビット 1, 入力バッファ 256 バイト).
OPEN "COM1:2400,N,8,1,CS0,DS0" FOR RANDOM AS #1 LEN = 256

DO            ' 通信メインループ.
    KeyInput$ = INKEY$                  'キーボードをチェックします.
    IF KeyInput$ = Quit$ THEN           ' [ESC] が押されたら,
        EXIT DO                         'ループを抜けます.
    ELSEIF KeyInput$ <> "" THEN         '何かキーが押されていれば,
        PRINT #1, KeyInput$;            'その文字をモデムに送ります.
    END IF

    'モデムをチェックします. 入力待ちの文字があれば (LOC(1) が 0 でな
    'ければ) それらを獲得して画面に表示します.
    IF LOC(1) THEN            'LOC(1) は入力待ちの文字数を返します.
        ModemInput$ = INPUT¥(LOC(1), #1)
        PRINT #2, ModemInput$;
    END IF
LOOP

CLOSE               '通信を終了します.
CLS
END
```

13 マウス，キーボード，ドラッグ & ドロップ イベント

Visual Basic のフォームを使うことの大きな利点の 1 つは、マウスやキーボードによるユーザーの操作は Visual Basic が自動的に監視していて、作成するアプリケーションでは、ユーザーの操作をイベントとして処理できることです。この章では、マウスやキーボードを通じてユーザーが行う操作を処理する方法について説明します。

フォームが表示されている場合、Visual Basic は、マウス操作とキーボード入力を監視して、ユーザーの操作が行われたとき、アプリケーション中の対応するイベント プロシージャを自動的に呼び出します。例えば、フォーム上でマウスがクリックされたことに応答する処理は、Form_Click イベント プロシージャに記述しておけば、ユーザーがその操作を行ったときに自動的に呼び出されて実行されます。通常のアプリケーションでは、マウス操作の制御は Click (クリック) と DblClick (ダブルクリック) イベントだけで処理できますが、より詳細な処理を行いたいときは、MouseDown と MouseUp の 2 つのイベントを利用します。また、MouseMove イベントを使うと、マウス ボタンの状態にかかわらず、マウス ポインタが移動するたびに特定の処理を行うことができます。同様に、ユーザーがキーボードのキーを押して離すと、KeyDown，KeyUp，KeyPress の各イベントが発生します。さらに Visual Basic では、ドラッグ & ドロップのための処理も記述できます。ドラッグ & ドロップとは、マウス ボタンを押してコントロールを選択し、それを別の位置までドラッグしてドロップする (マウス ボタンを離す) 操作です。

フォームを使用するアプリケーションでも、フォームが表示されていないときは、**INKEY$** や **INPUT** などの従来の Basic のステートメントを使ってキーボード入力を処理します。また、従来の Basic にはマウスの操作を直接処理するためのステートメントは用意されていませんが、マウス ツール キットを利用すると、フォームが表示されていないときでも、アプリケーション中からマウスの操作を監視することができます。マウス ツール キットについては、この章の最後に説明します。

13.1 マウス イベント

フォーム，ピクチャー ボックス，ラベル，リスト ボックス，コンボ ボックスと、すべてのファイル システム コントロールは、次に示す各イベントを認識可能です。これらのイベントを、マウス イベントと呼びます。

イベント	説明
MouseDown	ユーザーがいずれかのマウス ボタンを押したときに発生する。
MouseUp	ユーザーがいずれかのマウス ボタンを離したときに発生する。
MouseMove	マウス ポインタが画面上を移動するたびに発生する。

コントロール上でこれらの操作が行われたときは、そのときマウス ポインタの下になっているコントロールに対して各イベントが発生します。フォーム上のコントロールが配置されていない場所でこれらの操作が行われたときは、フォームに対して発生します。

ユーザーがマウスのボタンを押してから離すまでは、同じコントロールまたはフォームに対して続けてイベントが発生します。例えば、あるコントロールの上でユーザーがマウスのボタンを押し、押したままマウスを移動して離した場合でも、離したときのマウス ポインタの位置にかかわらず、同じコントロールが MouseDown イベントと MouseUp イベントの両方を認識します。

マウス イベントに対するイベント プロシージャは、次の形式で記述します。

```
SUB ctlname_MouseDown (Button AS INTEGER, Shift AS INTEGER, X
 ⇒ AS SINGLE, Y AS SINGLE)

SUB ctlname_MouseUp (Button AS INTEGER, Shift AS INTEGER, X AS
 ⇒ SINGLE, Y AS SINGLE)

SUB ctlname_MouseMove (Button AS INTEGER, Shift AS INTEGER, X
 ⇒ AS SINGLE, Y AS SINGLE)
```

それぞれのイベント プロシージャで、引数の意味は共通です。次のようになっています。

引数	説明
Button	下位 2 ビットがマウス ボタンの状態を表す整数。
Shift	下位 3 ビットが [SHIFT]、[CTRL]、[メニュー] の 3 つのキーの状態を表す整数。
X, *Y*	イベントが発生したオブジェクトの左上角を (0,0) とした、マウス ポインタの相対位置を示す整数。

Button と *Shift* の 2 つの引数は、ビットごとに別の意味を持っています。このような引数を、“ビット フィールド引数” と呼びます。詳しくは、「*Button* 引数と *Shift* 引数」で説明します。

マウスに関するイベントの発生と認識

マウス ポインタは、フォームやコントロールを含む、デスクトップ上 (画面全体) のどの位置にでも移動できます。ただし、コントロールやフォームが表示されていない場所でユーザーがマウスを操作しても、イベントは発生しません。フォームが表示されていなければ、マウス ポインタも表示されません。ここでは、タイマー コントロールを 1 つだけ持つ、Mouser という名前のフォームを作成し、この動作を確認してみます。コントロールの名前は、デフォルトの Timer1 に設定します。また、タイマーのインターバル (Interval プロパティ) は 900 に設定してください。次の 2 つのプロシージャを記述します。

```
SUB Form_DblClick
    HIDE
END SUB

SUB Timer1_Timer
    SHOW
END SUB
```

実行すると、フォームが表示されます。このフォームをダブル クリックすると、フォームは画面から消えますが、すぐに Timer1_Timer イベントによって再表示されます。フォームが表示されていない間は、マウス ポインタも表示されず、マウスに関するイベントは発生しません。フォームが再び表示されると、フォームが消える前と同じ位置にマウス ポインタも表示されます。

マウスの移動と座標

コントロールが配置されていないフォームが表示されているとき、マウス ポインタが移動すると、Visual Basic はマウス ポインタの位置をフォームに対する相対位置として追跡します。マウスの位置が変わると、MouseMove イベントが発生します。MouseMove イベントに対するイベント プロシージャには、マウス ポインタの座標が、引数 X と Y として渡されます。前に作成したフォーム Mouser に次のようなイベント プロシージャ Form_MouseMove を追加すると、フォームが表示されているときにマウスがどのように追跡されるのかを確かめることができます。

```
SUB Form_MouseMove (Button AS INTEGER, Shift AS INTEGER, X AS
 ⇒ SINGLE, Y AS SINGLE)
    IF X >= 0 AND X < ScaleWidth AND Y >= 0 AND Y < ScaleHeight
    ⇒ THEN
    'マウス ポインタがフォーム上にあるときは、マウス ポインタの座標と、
    'フォーム上にあることを示すメッセージを表示します。
        Caption = STR$(Y) + ", " + STR$(X) + " フォームのクライア
       ⇒ ント領域内です"
    ELSE
    'マウス ポインタがフォーム上にないときは、マウス ポインタの座標と、
    'フォーム上にないことを示すメッセージを表示します。
        Caption = STR$(Y) + ", " + STR$(X) + " フォームのクライア
       ⇒ ント領域外です。"
    END IF
END SUB
```

このプロシージャは、マウス ポインタの座標をフォームのキャプションとして表示します。画面上でマウスを動かしてみてください。実行開始直後には、フォームのキャプションには座標は表示されません。マウス ポインタがフォームの内側に移動してくると、座標が表示されます。その後、マウスのボタンを押していなければ、フォームの外や境界線上にマウス ポインタがあるときは、座標の表示が変わらないはずです。また、キャプションとして表示されるメッセージは、常に“領域内”となっているはずです。

MouseMove イベントが発生する部分、つまりフォームの内側を、そのフォームの“クライアント領域”と呼びます。クライアント領域には、タイトル バーや最大化/最小化ボタンなど、フォームの境界線は含まれないことに注意してください。

IF ステートメントは、マウス ポインタがフォームのクライアント領域内にあるかどうかを判断します。マウス ポインタがフォーム上にあるときにマウス ボタンを押し、そのままドラッグすると、マウス ポインタがフォームの外側に出てからもマウスの座標は追跡されます。このときも、マウスの位置は、フォームのクライアント領域の左上角を (0, 0) とする相対座標で示されます。マウス ボタンを押したまま、マウス ポインタをフォームの上側に動かしたときは、Y 座標は負になります。同様に、マウス ポインタをフォームより左側に動かしたときは、X 座標は負になります。マウス ポインタがフォームの外にあるときにマウス ボタンを離すと、追跡は中止されます。IF ステートメントでは、ScaleWidth と ScaleHeight プロパティを使ってフォームの右側と下側の判断を行っています。これらのプロパティは、表示されているフォームのクライアント領域の右端と下端の座標を示します。

マウス ポインタ

マウス ポインタの形状は、MousePointer プロパティに値を代入することによって、12 種類の中から選択することができます。ただし個々のマウス ポインタの形状は、そのときのスクリーン モードによって若干の違いがあります。MousePointer プロパティに設定可能な値と対応する形状を次の表に示します。

MousePointer プロパティの値	形状
0	矢印 (デフォルト)
1	小型矢印
2	クロス
3	I ビーム
4	二重四角
5	サイズ (四方矢印)
6	斜め矢印 (右上から左下)
7	上下サイズ (上下矢印)
8	斜め矢印 (左上から右下)
9	左右サイズ (左右矢印)
10	上矢印
11	砂時計
12	禁止マーク

例えば、次のような Form_MouseMove プロシージャを記述すると、マウス ポインタが縦に動いたときと横や斜めに動いたときでマウス ポインタの形状を変化させることができます。**STATIC** 変数 oldX と oldY は、直前のマウス ポインタの位置を保持しています。

```
SUB Form_MouseMove (Button AS INTEGER, Shift AS INTEGER, X AS
 ⇒ SINGLE, Y AS SINGLE)
STATIC oldX AS INTEGER, oldY AS INTEGER
                                    '古い座標を格納するための変数
    IF X <> oldX THEN               'マウスが横や斜めに動いているとき
        MousePointer = 9
    END IF                          '縦に動いているとき
```

```
        MousePointer = 7
    END IF
    oldX = X: oldY = Y
END SUB
```

MousePointer プロパティは、ほとんどのコントロールと、フォーム，**SCREEN** オブジェクトに対して用意されています。この例では、フォームの MousePointer プロパティを変更していることに注意してください。フォーム上でマウス ポインタの形状を変更しても、マウス ボタンを離したままマウス ポインタをフォームの外に移動すると、自動的に元の形状に戻ります。

また、フォームに配置されているコントロールで独自の MousePointer プロパティを設定していれば、マウス ポインタがそのコントロール上に移動すると、形状が変化します。設定されていなければ、そのコントロールの親オブジェクト (通常はフォーム) の MousePointer プロパティの設定に従います。

そのときのアプリケーションの状況をユーザーに知らせたいとき、マウス ポインタの形状を変える便利です。例えば、あるコントロールが特別な動作をすることを示すため、マウス ポインタがそのコントロール上に移動したときだけ形状を変更することがあります。また、時間のかかる処理を実行しているとき、ユーザーが別の操作を行ってしまわないように、処理が終わるまでマウス ポインタの形状を砂時計にしておくこともよく行われます。

13.2 マウス イベントの使い方

ここでは、MouseMove, MouseUp, MouseDown の 3 つのイベントの具体的な使い方について説明します。まず、簡単な線を描くプログラムでマウスの動きの基本を説明した後、*Button* 引数と *Shift* 引数について説明します。

マウスの動きの追跡

前の説明の中で、マウス ポインタの移動方向を調べてマウス ポインタの形状を変更するプログラムを作りました。MouseMove イベントは、マウス ポインタの位置が変わるたびに発生するので、次のようなプログラムを使うと、コントロールを持たないフォーム上でのマウス ポインタの動きを簡単に追跡できます。

```
SUB Form_MouseMove (Button AS INTEGER, Shift AS INTEGER, X AS
 ⇒ SINGLE, Y AS SINGLE)
    'マウスがフォームのクライアント領域内にあるかどうか確認します。
    IF X >= 0 AND X < ScaleWidth AND Y >= 0 AND Y < ScaleHeight
    ⇒ THEN
        CurrentX = X: CurrentY = Y        'PRINT メソッドのための位置
        PRINT "#"                         'を設定して、現在位置に文字を
                                          '表示します。
    END IF
END SUB
```

マウス ポインタの位置が、フォームのクライアント領域内であるかどうかをチェックしていることに注意してください。このチェックを行わないと、ドラッグしたままマウス ポインタがフォームの上または左側に移動したとき、CurrentX または CurrentY プロパティに負の値が代入されてしまい、“プロパティの値が無効です”というエラーが発生します。

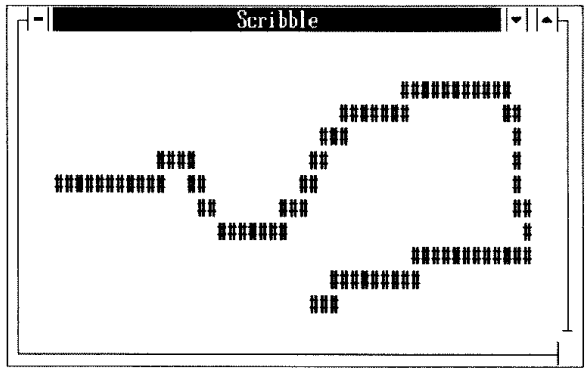


図 13.1　マウスの動きの追跡

MouseMove の制限

MS-DOS やマウス ドライバがマウスの動きを検出できるスピードには限度があります。このスピードより速くマウス ポインタを動かした場合、MouseMove イベントが発生する場所が連続しないことがあります。

MouseMove イベントは、頻繁に発生することに加えて、Visual Basic が処理を行うための負荷も小さくありません。したがって、MouseMove イベントに対するイベント プロシージャに、複雑で時間がかかる処理を記述することは避けてください。

13.3　*Button* 引数と *Shift* 引数

MouseDown, MouseUp, MouseMove の各イベントに対するイベント プロシージャには、*Button* 引数と *Shift* 引数が渡されます。これらの引数によって、左右どちらのマウス ボタンが押されたのか、また、[SHIFT], [CTRL], [メニュー] キーが同時に押されているかどうかをチェックすることができます。このような処理を記述しておけば、ユーザーはマウスだけでさまざまな操作を行うことができます。

Button 引数と *Shift* 引数は、どちらもビット フィールドになっています。ビット フィールドとは、各ビットがそれぞれ独立した意味を持つ数値です。*Button* 引数と *Shift* 引数は、どちらも整数 (**INTEGER**) としてイベント プロシージャに渡されます。このとき、その値を 2 進数で考えたときの特定の桁 (ビット) が、それぞれのボタンやキーの状態を表しています。

マウスのボタンの状態の検出

Button 引数は、ユーザーが左右どちらのボタンを操作しているのかを示します。次のように、下位 2 ビットが左右それぞれのマウス ボタンの状態を表します。

ビット	7	6	5	4	3	2	1	0
意味	未使用	未使用	未使用	未使用	未使用	未使用	右ボタン	左ボタン

MouseMove イベントと、MouseDown および MouseUp イベントで、*Button* 引数の扱いには違いがあります。

MouseDown イベントと MouseUp イベントでの *Button* 引数の使い方

MouseDown イベントで *Button* 引数 を使うと、どのボタンが押されたかを判断できます。また、 MouseUp イベントで *Button* 引数を使うと、どのボタンが離されたかを判断できます。どちらのイベントも、左右どちらのボタンが押されても発生するため、2 つのボタンが同時に押されたまたは離されたかどうかを調べることはできません。ユーザーが 2 つのボタンを同時に押すか離すかすると、*Button* 引数の値が異なるイベントが 2 回発生します。

注 意 MouseMove イベントでは、マウス ポインタの移動中に押されているボタンの状態をすべて調べることができます。つまり 2 つのボタンが同時に押されているかどうかも *Button* 引数を使って判断できます。詳しくは、「MouseMove イベントでの *Button* 引数の使い方」を参照してください。

MouseDown および MouseUp イベントでは、*Button* 引数は次の 2 つの値のどちらかになります。

値	意味
1 (2 進数でビット 0 だけが 1)	左ボタンによりイベントが発生。
2 (2 進数でビット 1 だけが 1)	右ボタンによりイベントが発生。

どちらのボタンの操作によって MouseDown または MouseUp イベントが発生したのかは、簡単なプログラムで調べることができます。次のプロシージャは、*Button* 引数が 1 か 2 かを調べ、メッセージを表示します。

```
SUB Form_MouseDown (Button AS INTEGER, Shift AS INTEGER, X AS
 ⇒ SINGLE, Y AS SINGLE)
    IF Button = 1 THEN PRINT "マウスの左ボタンが押されました。"
    IF Button = 2 THEN PRINT "マウスの右ボタンが押されました。"
END SUB
```

ユーザーが 2 つのボタンを同時に押したときは、MouseDown イベントが 2 回発生します。まず、少しでも先に押されたボタンに対するイベントが発生し、すぐにもう一方のボタンに対するイベントが発生します。同じように、ユーザーが 2 つのボタンを同時に離したときも、2 回の MouseUp イベントが発生します。

MouseMove イベントでの *Button* 引数 の使い方

MouseMove イベントでは、*Button* 引数は次の 4 つの値のいずれかになります。

値	意味
0	どちらのボタンも押されていない。
1 (2 進数でビット 0 だけが 1)	左ボタンだけが押されている。
2 (2 進数でビット 1 だけが 1)	右ボタンだけが押されている。
3 (2 進数でビット 0 と 1 の両方が 1)	両方のボタンが押されている。

次のプログラムは、左ボタンと右ボタンが同時に押され続けているときは何も表示しません。

```
SUB Form_MouseMove (Button AS INTEGER, Shift AS INTEGER, X AS
 ⇒ SINGLE, Y AS SINGLE)
    IF Button = 1 THEN PRINT "マウスの左ボタンだけが押されています。"
END SUB
```

どちらか片方のボタンが押されているかどうかだけを調べて、もう一方のボタンの状態を無視したいときは、**AND** 演算子を使います。次のプログラムは、押された各ボタンについてメッセージを表示し、他のボタンが押されているかどうかは無視します。

```
SUB Form_MouseMove (Button AS INTEGER, Shift AS INTEGER, X AS
 ⇒ SINGLE, Y AS SINGLE)
    IF Button AND 1 THEN PRINT "マウスの左ボタンが押されています。"
    IF Button AND 2 THEN PRINT "マウスの右ボタンが押されています。"
END SUB
```

AND 演算子はビット演算子であるため、式 Button AND 1 は、*Button* 引数のビット 0 が 1 であれば真 (0 以外) と評価されます。この式では、他のビットは意味を持ちません。また、式 Button AND 2 も、他のビットにかかわらず、*Button* 引数のビット 1 が 1 であれば真 (0 以外) と評価されます。

右，左，両方の 3 つの条件に分けたいときは、次のように単に等しいかどうかを調べます。

```
SUB Form_MouseMove (Button AS INTEGER, Shift AS INTEGER, X AS
 ⇒ SINGLE, Y AS SINGLE)
    IF Button = 1 THEN PRINT "マウスの左ボタンだけが押されています。"
    IF Button = 2 THEN PRINT "マウスの右ボタンだけが押されています。"
    IF Button = 3 THEN PRINT "マウスの両方のボタンが押されています。"
END SUB
```

次のように、検出するボタンのための定数を定義すれば、プログラムが読みやすくなります。

```
CONST LEFT_BUTTON = 1
CONST RIGHT_BUTTON = 2
```

[SHIFT]，[CTRL]，[メニュー] キーの状態の検出

Shift 引数は、マウスの操作が行われたときに、[SHIFT]，[CTRL]，[メニュー] の 3 つのキーが押されていたかどうかを示します。次のように、下位 3 ビットだけが意味を持ちます。

ビット	7	6	5	4	3	2	1	0
意味	未使用	未使用	未使用	未使用	未使用	[メニュー]	[CTRL]	[SHIFT]

[SHIFT]，[CTRL]，[メニュー] の各キーの状態によって、*Shift* 引数の各ビットがセットされます。次の 8 種類のパターンがあります。

値	意味
0	[SHIFT], [CTRL], [メニュー] キーのどれも押されていない。
1 (2 進数 001)	[SHIFT] キーが押されている。
2 (2 進数 010)	[CTRL] キーが押されている。
4 (2 進数 100)	[メニュー] キーが押されている。
3 (2 進数 011)	[SHIFT] キーと [CTRL] キーが押されていて、[メニュー] キーは押されていない。
5 (2 進数 101)	[SHIFT] キーと [メニュー] キーが押されていて、[CTRL] キーは押されていない。
6 (2 進数 110)	[CTRL] キーと [メニュー] キーが押されていて、[SHIFT] キーは押されていない。
7 (2 進数 111)	[SHIFT], [CTRL], [メニュー] キーが押されている。

次のようなイベント プロシージャを記述すると、マウスが移動しているときにどのキーが押されているかを表示できます。この場合、例えば 3 つのキーがすべて押されているときは、すべての条件が真になり、7 つのメッセージが表示されます。

```
SUB Form_MouseDown (Button AS INTEGER, Shift AS INTEGER, X AS
 ⇒ SINGLE, Y AS SINGLE)
    IF Shift AND 1 THEN PRINT "[SHIFT] キーが押されています。"
    IF Shift AND 2 THEN PRINT "[CTRL] キーが押されています。"
    IF Shift AND 4 THEN PRINT "[メニュー] キーが押されています。"
    IF (Shift AND 3) = 3 THEN PRINT "[SHIFT] と [CTRL] キーが押
     ⇒ されています。"
    IF (Shift AND 5) = 5 THEN PRINT "[SHIFT] と [メニュー] キーが押さ
     ⇒ れています。"
    IF (Shift AND 6) = 6 THEN PRINT "[CTRL] と [メニュー] キーが押さ
     ⇒ れています。"
    IF (Shift AND 7) = 7 THEN PRINT "[SHIFT], [CTRL], [メニュー] の
     ⇒ 3 つのキーが押されています。"
END SUB
```

複数のビット (キーの状態) の複合条件を調べる場合、**AND** 演算子と = 演算子を組み合わせて使います。まず、ビット フィールド (この場合は Shift) と複数の特定のビットの **AND** を取り、他のビットの状態を無効にします。この操作を“マスクする”と言います。例えば、[メニュー] キーが押されているかどうかにかかわらず、[SHIFT] キーと [CTRL] キーが同時に押されていることを調べるには、*Shift* 引数の値と 3 ([SHIFT] を示す 1 +

[CTRL] を示す 2) の **AND** を取り、[メニュー] の状態を無効にします。その後、結果の値が 3 であるかどうかを調べます。

例えば、[メニュー] キーの状態にかかわらず、[SHIFT] と [CTRL] の 2 つのキーが押されていることを調べるとき、次の 2 つの **IF** ステートメントは、どちらも条件判断として正しく機能しません。

```
IF Shift AND 3 THEN
IF (Shift AND 1) AND (Shift AND 2) THEN
```

Shift AND 3 は、*Shift* 引数が 1, 2, 3 のどの場合でも真 (0 以外) と評価されてしまいます。つまり [SHIFT] と [CTRL] のどちらか片方のキーだけが押されていれば真になり、両方のキーに対するチェックにはなりません。また 2 つ目の **IF** ステートメントでは、Shift AND 1 は、両方のキーが押されていたら 0 以外に評価されます (3 AND 1 = 1)。同様に、Shift AND 2 も 0 以外に評価されます (3 AND 2 = 2)。しかし、この 2 つの評価結果の **AND** を取ると (1 AND 2)、その結果は 0 になります。つまり、両方のキーが押されていたら偽になってしまいます。

13.4　マウス イベントの例

ここでは、マウスの動きに従ってピクチャー ボックス上に文字で線を描く、LineDraw アプリケーションを作成します。まず、新しいフォームを作成し、フォームの半分以上の大きさのピクチャー ボックス (CtlName = Picture1) をフォームの左上隅に配置してください。

オブジェクト	プロパティの設定	使用するイベント
ピクチャー ボックス	CtlName = Picture1 AutoRedraw = True	Picture1_MouseDown Picture1_MouseUp Picture1_MouseMove

ピクチャー ボックスの AutoRedraw プロパティは、True に設定してください。これによって、フォームのサイズ変更などの再描画が必要な動作を行ったとき、進行中の作業をむだにせずに済みます。前の例では、文字を使って線を描画するためのプログラムを Form_MouseMove イベント プロシージャの中に記述しました。LineDraw アプリケーションでは、描画はピクチャー ボックスの中にだけ行うため、描画のためのコードは Picture1_MouseMove に記述します。

Button 引数 と *Shift* 引数の使い方の例

マウス ボタンが押されているときだけ線が描画されるように設計すると、前の例よりもずっと便利なプログラムになります。Picture1_MouseMove イベント プロシージャの中で *Button* 引数を調べれば、この機能を実現できます。ここでは、ユーザーがマウスの左ボタンを押しているときだけ、線が引かれるようにします。右ボタンが押されているかどうかはチェックしません。

また、*Shift* 引数を使って、[CTRL] キーを押しながらマウスを動かしたら、すでに引いた線を消す機能を加えてみます。線を消すには、そのときの出力位置 (CurrentX および CurrentY プロパティが示す座標) にスペースを出力します。

Picture1_MouseMove プロシージャは、次のようになります。

```
SUB Picture1_MouseMove (Button AS INTEGER, Shift AS INTEGER, X
 ⇒ AS SINGLE, Y AS SINGLE)
    'マウス ポインタが描画領域内にないときは、すぐにプロシージャを抜けます。
    IF X < 0 OR X > Picture1.ScaleWidth THEN EXIT SUB
    IF Y < 0 OR Y > Picture1.ScaleHeight THEN EXIT SUB
    IF Button AND 1 THEN     '左ボタンが押されているかどうか確認します。
        Picture1.CurrentX = X
        Picture1.CurrentY = Y
        IF Shift AND 2 THEN          '[CTRL] キーが押されているときは
            Picture1.PRINT " "       'スペースを表示し、
        ELSE                         '押されていないときは
            Picture1.PRINT "#"       '描画文字を表示します。
        END IF
    END IF
END SUB
```

MouseUp イベントと MouseDown イベントを使った例

このプロシージャは MouseMove イベントに対応しているため、左ボタンが押されたときの位置には文字が表示されません。そこで、マウスの左ボタンを押した位置から描画が始まるように、MouseDown イベントにも対応するプロシージャを記述してみます。また、描画をもっと簡単にするための拡張も合わせて行います。

ここでは、LineDraw アプリケーションに 2 つの変更を加えます。まず、ある 2 点を指定して簡単に直線を引けるようにします。マウスの右ボタンである 1 点をクリックし、次に別の 1 点をクリックすると、2 点が水平または垂直線上に並んでいれば、その 2 点間に直線が描かれます。次に、マウスの左ボタンを押したところから線が描かれるようにします。改良した LineDraw アプリケーションでは、次の 3 つのマウス操作すべてに応答するようにしなければなりません。

- マウス ボタンが押された (MouseDown)
- マウス ボタンが離された (MouseUp)
- マウス ポインタが移動した (MouseMove)

クリックによる線の描画

MouseDown イベントと MouseUp イベントを使えば、ちょうどマウスの左ボタンが押された位置から線を引き始め、ボタンが離された位置で終わる処理を記述できます。複数のイベント プロシージャで共有するグローバル変数を使えば、そのときの状態を示す情報のやりとりが行えます。まず、モジュール レベルに次のように記述して、グローバルな定数と変数を宣言します。

```
CONST FALSE = 0, TRUE = NOT FALSE
DIM SHARED lineFlag AS INTEGER
```

lineFlag 変数は、右ボタンを使った直線の描画のモードであるかどうかを示すフラグです。False (0) であれば通常の描画のモードであることを示し、True (-1) であれば、すでにある 1 点が右ボタンでクリックされ、次のクリックを待っている状態、つまり直線の描画のモードであることを意味します。

Picture1_MouseDown には、**STATIC** 変数 oldX と oldY も使われています。この2つの変数は、最初にマウスの右ボタンが押されたときの座標を保持しておくために使われます。

右ボタンと左ボタンを区別するため、条件判断に **AND** ではなく = を使っていることに注意してください。

```
SUB Picture1_MouseDown (Button AS INTEGER, Shift AS INTEGER, X
 ⇒ AS SINGLE, Y AS SINGLE)
    STATIC oldX AS INTEGER, oldY AS INTEGER
                                        '座標を保存しておくための変数
    'マウス ポインタが描画領域内にないときは、すぐにプロシージャを抜けます。
    IF X < 0 OR X > Picture1.ScaleWidth THEN EXIT SUB
    IF Y < 0 OR Y > Picture1.ScaleHeight THEN EXIT SUB
    Picture1.CurrentX = X
    Picture1.CurrentY = Y           '文字を出力する位置を設定します.

    IF Button = 1 THEN              '左ボタンが押されているとき:
        lineFlag = FALSE            '描画または消去を開始します。
        Picture1.MousePointer = 2
                'マウス ポインタの形を線の描画を表す十字に設定します。
        IF Shift AND 2 THEN         '[CTRL] キーが押されているときは
            Picture1.PRINT " "      'スペースを表示し、
        ELSE                        '押されていないときは
            Picture1.PRINT "#"      '描画文字を表示します。
        END IF

    ELSE                            '右ボタンが押されているとき:
        IF lineFlag <> TRUE THEN    'この場合は描画開始点です。
            Picture1.MousePointer = 4
                '線のクリックであることを示すためにマウス ポインタを
                '二重四角に設定します。
            Picture1.PRINT "*"      '最初のクリックの場所に
            oldX = X                'アスタリスクを表示して、
            oldY = Y                'モジュール レベルの SHARED
            lineFlag = TRUE         'フラグをセットします。
        ELSE
        'この場合は線のクリックの 2 番目の場所です。したがって描画開始点
        'から線を引きます。
            CALL ClickLine(Shift, oldX, oldY, X, Y)
            lineFlag = FALSE        '線のクリックのフラグをオフに
                                    'します。
        END IF
    END IF
END SUB
```

MouseUp イベント プロシージャの内容は非常に簡単です。直線が引けたら、マウス ポインタをデフォルトの形状に戻し、直線描画中ではないことを示します。

```
SUB Picture1_MouseUp (Button AS INTEGER, Shift AS INTEGER, X AS
 ⇒ SINGLE, Y AS SINGLE)
    IF lineFlag = FALSE THEN Picture1.MousePointer = 0
END SUB
```

ClickLine **SUB** プロシージャでは、まずクリックされた 2 点の座標が水平または垂直線上にあるかどうかを確認し、描画方向を決めます。次に、水平線か垂直線かにより、2 つの **FOR** ループのどちらかで線を引きます。[CTRL] キーが押されているときは、スペースを出力して線を消します。[CTRL] キーが押されていないときは、前の座標から新しい座標まで線を引きます。

```
SUB ClickLine (Shift AS INTEGER, oldX AS INTEGER, oldY AS
 ⇒ INTEGER, X AS SINGLE, Y AS SINGLE)
    IF oldX <> X AND oldY <> Y THEN      '斜めの線は描画できません。
        Picture1.CurrentX = oldX
        Picture1.CurrentY = oldY
        Picture1.PRINT " "       '目印のアスタリスクを消します。
        EXIT SUB
    END IF
    IF oldX > X OR oldY > Y THEN dirc% = -1 ELSE dirc% = 1
    'X が変化している場合、水平の線を引きます。
    IF oldX <> X THEN
        FOR ind% = oldX TO X STEP dirc%
            Picture1.CurrentX = ind%
            Picture1.CurrentY = Y
            IF Shift = 2 THEN
                Picture1.PRINT " "
            ELSE
                Picture1.PRINT "#"
            END IF
        NEXT ind%
    END IF
    'Y が変化している場合、垂直の線を引きます。
    IF OldY <> Y THEN
        FOR ind% = OldY TO Y STEP dirc%
            Picture1.CurrentX = X
            Picture1.CurrentY = ind%
            IF Shift = 2 THEN
                Picture1.PRINT " "
            ELSE
                Picture1.PRINT "#"
            END IF
        NEXT ind%
    END IF
END SUB
```

改良した LineDraw アプリケーションでは、イベントの動作間の関係が理解できます。ユーザーがマウスの左ボタンを押すと、まず MouseDown イベントが発生します。Picture1_MouseDown イベント プロシージャにより、最初の文字 (線の開始位置) が表示または消去されます。そのあと、マウスがドラッグされると、MouseMove が連続して呼び出されます。ユーザーがマウスの右ボタンをクリックすると、lineFlag がセットされ、開始点の座標が保存されて、もう 1 度右ボタンがクリックされると、直線が描画されます。

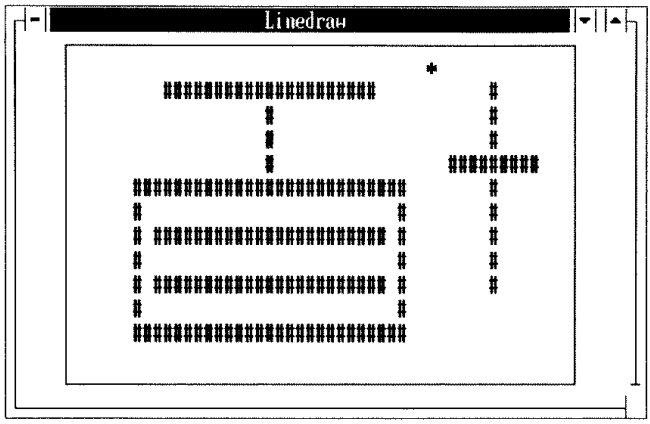


図 13.2　LineDraw アプリケーション

アプリケーションを設計するとき、関連するイベントの意味と、それに対するアプリケーションの応答をメモしておくと役に立ちます。このアプリケーションで利用した 3 つのイベントはすべてマウスに関連したイベントで、次のような意味を持っています。

- Picture1_MouseDown　どちらのマウス ボタンかによって動作が異なる。

  左ボタン:　線の最初の点の描画または消去
  右ボタン:　直線の描画の開始または終了

- Picture1_MouseUp　マウス ポインタの形状を戻す。
- Picture1_MouseMove　左ボタンなら、線を描画または消去する。

次のような定数をプログラムのモジュール レベルで定義すれば、プログラムが読みやすくなります。

```
CONST SHIFT_MASK = 1
CONST CTRL_MASK= 2
CONST ALT_MASK= 4
```

CONSTANT.BI ファイルには、このような定数が数多く定義されています。第 6 章「プロジェクトの管理」で説明したように、必要に応じてこれらの定数を自分のプログラムにコピーして使うことができます。また、メタコマンド **$INCLUDE** でファイル全体をインクルードして使うこともできます。

13.5　キーボード イベント

キーボードに関するイベントには、KeyDown, KeyUp, KeyPress の 3 種類があります。ユーザーがあるキーを押すと、KeyDown, KeyPress , KeyUp の順でイベントが発生します。KeyDown と KeyPress イベントは、どちらもユーザーがキーを押したときに発生しますが、ファンクション キーなどのいくつかの特殊キーに対しては、KeyPress イベントは発生しません。

ユーザーがキーを押したとき、そのときフォーカスを持っているオブジェクトがイベントを認識します。フォームは、コントロールが何も配置されていないか、配置されているコントロールがすべて無効になっているときだけ、フォーカスを持つことができます。

注 意 ユーザーがあるキーを押し続けたとき、自動的にキーがリピートされ、同じキーを何度も押し続けたときと同じ結果になります。

ユーザーが、[SHIFT], [CTRL], [メニュー] 以外のキーを押したまま何かほかのキーを押すと、最初のキーが離されていなくても、後で押されたキーの KeyDown イベントが発生します。そのあとで最初に押されたキーが離されても、すでにキーは離されたと認識しているため、KeyUp イベントは発生しません。この状態を“ロールオーバー”と呼びます。

KeyPress イベント

KeyPress イベントは、ある文字が入力されたときに発生します。イベント プロシージャに渡された引数の値によって、入力された文字を判断できます。3 つのキーボード イベントの中では、最も簡単に使えます。KeyPress イベント プロシージャの書式は、次のとおりです。

```
SUB CtlName_KeyPress (KeyAscii AS INTEGER)
```

引数 *KeyAscii* には、押されたキーの文字コード (ASCII コードまたは漢字連続コード) が格納されます。例えば、[CAPS LOCK] キーがロックされていない状態で単独の [L] キーが押されると、*KeyAscii* の値は小文字の "l" の ASCII コードである 108 になります。また、[SHIFT] キーと共に [L] キーが押されたときは、*KeyAscii* の値は大文字の "L" の ASCII コードである 76 になります。つまり KeyPress イベントは、キーボードの状態を取得するのではなく、押されたキーで生成された文字を返します。ユーザーが、方向キーや [CTRL] キー，[SHIFT] キー，[メニュー] キーなどの文字を生成しないキーだけを押しても、KeyPress イベントは発生しません。文字を生成するキーが押されたときは、KeyDown, KeyPress, KeyUp の順にイベントが発生します。[ENTER]，[ESC]，[DEL]，[SPACE]，[BS] などのキーは、画面に直接文字が表示されるわけではありませんが、特定の文字コードに割り当てられているため、KeyPress イベントが発生します。

日本語入力システムがオンになっている状態では、ユーザーからの入力が確定したときに KeyPress イベントが発生します。何文字かが 1 度に確定されたときは、1 文字に対して 1 度のイベントが続けて発生します。

KeyDown イベントと KeyUp イベント

KeyDown イベントと KeyUp イベントは、キーボードに応答するための最も低レベルのイベントです。この 2 つのイベントは、KeyPress イベントでは検出できない次のようなキーおよびキーの状態を検出する場合に使います。

- ファンクション キー
- [SHIFT]，[CTRL]，[メニュー] キーの組み合わせ

- 方向キー

  一部のコントロール (コマンド ボタン，オプション ボタン，チェック ボックス) は、方向キーをイベントとして認識せず、単に次のコントロールにフォーカスを移します。

- [PAGE UP] キーと [PAGE DOWN] キー
- テンキーの数字と通常のキーの数字の区別
- アクセス キーのうちメニュー コマンドに割り付けられていないキー

KeyDown イベントと KeyUp イベントを使った低レベルのキーボード処理ルーチンは、プログラミングが若干複雑になります。まず、行いたい処理を KeyPress イベントだけを使って記述できないか、十分に検討してください。一般に、KeyPress イベントを使った方が記述は簡単です。また、第 3 章「イベントドリブン アプリケーションの作成」説明されているショートカット キーやアクセス キーを使えば、特にキーボード イベントを使わなくてもキー入力を処理できます。通常、ショートカット キーが用意できるのはメニュー コマンドに対してだけですが、ショートカット キーが使えれば、ファンクション キーと [SHIFT] キーとの組み合わせも利用できます。

KeyUp, KeyDown イベントに対するイベント プロシージャの書式は次のとおりです。

```
SUB CtlName_KeyUp (KeyCode AS INTEGER, Shift AS INTEGER)
SUB CtlName_KeyDown (KeyCode AS INTEGER, Shift AS INTEGER)
```

引数 *KeyCode* は、押されたキーまたは離されたキーを示すコードです。*Shift* 引数 は、キーボード イベントの発生時の [SHIFT]，[CTRL]，[メニュー] の 3 つのキーの状態を返すビット フィールド引数です。

KeyCode 引数

イベント プロシージャには、*KeyCode* 引数として、押されたまたは離されたキーを示す整数が渡されます。アルファベットや数字，記号のキーが押されたときは、それぞれキートップに表示されている文字を示す ASCII コードが格納されます。特殊キーとコードの対応については、『機種依存情報』を参照してください。これらのコードを示す定数は、インクルード ファイル CONSTANT.BI で定義されています。このファイルから、必要な定義をモジュール レベルにコピーしてください。例えば、ファンクション キーに対しては次のように定義されています。

```
CONST KEY_F1 = 112
CONST KEY_F2 = 113
CONST KEY_F3 = 114
CONST KEY_F4 = 115
CONST KEY_F5 = 116
```

例えば、"1" が押されても "!" が押されても、*KeyCode* 引数としては "1" の ASCII コードである 49 が渡されます。一方、テンキー上の "1" が押されたときは、別のコードである 97 が渡されます。

次のプロシージャは、[F2] キーが押されるとアプリケーションを終了します。

```
SUB Text1_KeyDown (KeyCode AS INTEGER, Shift AS INTEGER)
    CONST KEY_F2 = 113
    IF KeyCode = KEY_F2 THEN
        SCREEN.HIDE
        END
    END IF
END SUB
```

Shift 引数

Shift 引数 はビット フィールド引数で、マウスに関するイベントの *Shift* 引数と同じ意味です。つまり、下位 3 ビットで [SHIFT]，[CTRL]，[メニュー] の 3 つのキーの状態を表します。

次のようにして、各シフト キーの状態を調べることができます。

```
ShiftDown% = (Shift AND SHIFTKEY) > 0
CtrlDown% = (Shift AND CTRL) > 0
AltDown% = (Shift AND ALT) > 0
```

次のプロシージャは、[SHIFT]+[メニュー]+[F2] キーが押されるとアプリケーションを終了します。

```
SUB Text1_KeyDown (KeyCode AS INTEGER, Shift AS INTEGER)
    CONST SHIFT_MASK = 1
    CONST ALT_MASK = 4
    CONST KEY_F2 = 113

    ShiftDown% = (Shift AND SHIFT_MASK) > 0
    AltDown% = (Shift AND ALT_MASK) > 0
    F2Down% = (KeyCode = KEY_F2)

    IF ShiftDown% AND AltDown% AND F2Down% THEN
        SCREEN.HIDE
        END
    END IF
END SUB
```

フォーム全体に有効な処理ルーチン

例えば、どのコントロールにフォーカスがあるときでも、ユーザーがあるキーを押したらアプリケーションが終了するようにするには、各イベント プロシージャから呼び出すための汎用的なジェネラル プロシージャを記述します。

次の例では、テキスト ボックスとピクチャー ボックスを持つフォームで、ユーザーが [F2] キーを押したときに実行を終了するために、KeyDownHandler というジェネラル プロシージャを使っています。2 つの KeyDown プロシージャは、引数 *KeyCode* と *Shift* を KeyDownHandler に引き渡しています。

```
SUB Text1_KeyDown (KeyCode AS INTEGER, Shift AS INTEGER)
    KeyDownHandler KeyCode, Shift
END SUB
```

```
SUB Picture1_KeyDown (KeyCode AS INTEGER, Shift AS INTEGER)
    KeyDownHandler KeyCode, Shift
END SUB

SUB KeyDownHandler (KeyCode AS INTEGER, Shift AS INTEGER)
    CONST KEY_F2 = 113
    IF KeyCode = KEY_F2 THEN
        SCREEN.HIDE
        END
    END IF
END SUB
```

13.6　ドラッグ & ドロップ

Visual Basic のアプリケーションを設計するとき、フォーム デザイナでは、フォーム上でコントロールをドラッグして位置や大きさを変更します。マウス ボタンを押したままコントロールを動かすことを“ドラッグ”、ドラッグ後にマウス ボタンを離すことを“ドロップ”といい、この機能をドラッグ & ドロップと呼びます。アプリケーションでドラッグ & ドロップの機能を利用すると、操作性を向上させることができます。

ドラッグ & ドロップは、複数のフォームにまたがって実行することもできます。アプリケーションを利用するユーザーは、他のフォーム上など、画面上のどこにでもコントロールをドラッグすることができます。どんなドラッグ操作に対しても応答するようにプログラムを設計することは可能ですが、この場合、設計には十分な注意が必要になります。

ユーザーがコントロールをドラッグすると、コントロールの形を示す黒い外枠が表示されます。この外枠は、マウス ポインタといっしょに移動します。次の図は、フォーム上に配置されたコマンド ボタンをドラッグしている状態を示しています。

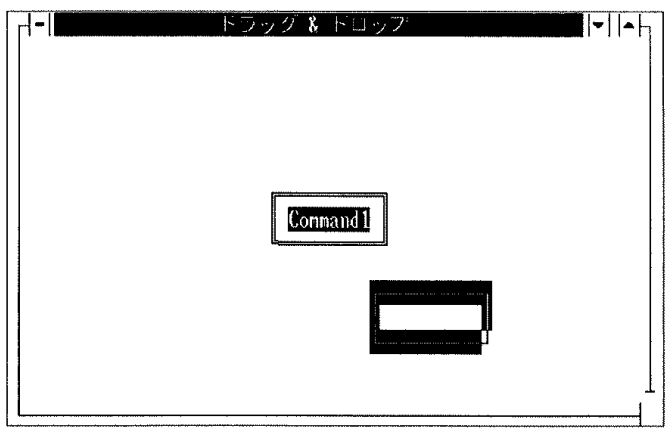


図 13.3　実行時におけるコントロールのドラッグ

アプリケーションの実行中にコントロールの位置を移動するには、その処理を行うコードを記述しなければなりません。次の表は、ドラッグ & ドロップに関連するプロパティ，イベント，メソッドを示します。

種類	項目	説明
プロパティ	DragMode	コントロールの自動ドラッグとマニュアルドラッグを切り替える。
イベント	DragDrop	コントロールがオブジェクト上にドロップされたときに発生する。
	DragOver	コントロールがドラッグされてオブジェクト上に重なったときに発生する。
メソッド	Drag	マニュアル ドラッグを開始または中止する。

メニューとタイマー以外のすべてのコントロールは、表に示したすべてのプロパティを持ち、実行時にドラッグすることができます。フォームは、DragDrop イベントと DragOver イベントを認識しますが、DragMode プロパティはありません。

自動ドラッグ モード

コントロールの DragMode プロパティを 1 (自動) に設定すると、コード中で特に処理を行わなくても、ユーザーがそのコントロールをドラッグできるようなります。

```
Thing.DragMode = 1
```

この設定により、ドラッグは常に有効になります。0 (マニュアル) に設定してドラッグを制御する方法については、あとで説明します。

注 意 DragMode プロパティを 1 に設定して自動ドラッグ モードにすると、ドラッグされているコントロールは、マウスに関連する Click や MouseMove などの他のイベントを認識しなくなります。

オブジェクトのドロップ

コントロールをドラッグしてからマウス ボタンを離すと、離された先の位置のオブジェクトに対して DragDrop イベントが発生します。このイベントには、いろいろな方法で応答することができます。コード中で特に指定を行わなければ、ドロップしてもコントロールは元の位置に配置されたままです。

ドラッグ & ドロップにおいて、ドラッグされるコントロールを "ソース" と呼びます。また、ドロップ先のフォームやコントロールを "ターゲット" と呼びます。ソースの対象は、メニューとタイマー以外のすべてのコントロールです。また、ドロップしたときに、ターゲットのオブジェクトに対して DragDrop イベントが発生します。

マウス ボタンが離されたとき、マウス ポインタがコントロールの境界内にあれば、そのコントロールがターゲットになります。また、マウス ポインタがフォーム上のコントロールが配置されていない部分にあれば、フォームがターゲットになります。

DragDrop イベントに対するイベント プロシージャには、*Source, X, Y* の 3 つの引数が渡されます。引数 *Source* は、ソース コントロールを示します。例えば、あるコント

ロールがフォームに対してドロップされると、次のような Form_DragDrop プロシージャが呼び出されます。

```
SUB Form_DragDrop (Source AS CONTROL, X AS SINGLE, Y AS SINGLE)
```

Source は、**CONTROL** と宣言されていることに注意してください。イベント プロシージャ内では、*Source* パラメータに指定した名前で、ドロップされたコントロールのプロパティを操作したりメソッドを使用することができます。

ターゲットのイベント プロシージャに対して、ソースに関する情報を伝えたいときは、ソースのコントロールの Tag プロパティを使います。Tag プロパティは、他の動作に影響を与えずに、任意の文字列を代入できる便利なプロパティです。例えば、CtlName プロパティと同じ値を Tag プロパティにあらかじめ設定しておけば、ターゲットのイベント プロシージャで、ソースがどのコントロールであるかを判断できます。

Source パラメータには必ずコントロールが引き渡されますが、さまざまな種類のコントロールが渡される可能性があります。例えば、ドロップされたコントロールがテキスト ボックスのとき、Source.Value を参照しようとすると、ランタイム エラーが発生します。これは、テキスト ボックスには Value プロパティがないためです。

IF...THEN...ELSE ステートメントで **TYPEOF** キーワードを使うと、ドロップされたコントロールの種類を調べることができます。

```
IF TYPEOF Source IS PictureBox THEN
    Source.Backcolor = 5
ELSE
    Source.Backcolor = 3
END IF
```

コントロールの種類を調べる必要がない場合もあります。例えば、ドラッグ可能なすべてのコントロールは、必ず Visible プロパティを持っています。これを利用して、コントロールが特定のコントロールの上にドロップされたら、そのコントロールを非表示にするというような処理が可能です。次のイベント プロシージャは、コントロールが T というピクチャー ボックスにドロップされたとき、そのコントロールを非表示にします。

```
SUB T_DragDrop (Source AS CONTROL, X AS SINGLE, Y AS SINGLE)
    Source.Visible = 0
END SUB
```

ドラッグ時のオブジェクトの強調表示

あるオブジェクトの上にコントロールがドラッグされて来たとき、そのオブジェクトの表示方法を変えると、ユーザーにさまざまな情報を伝えることができます。例えば、ドロップ可能なオブジェクトの上に来たらそのオブジェクトを明るい色に変えるようにすれば、ユーザーはドロップできることが分かります。

ドラッグされたコントロールが、あるオブジェクトに重なると、そのオブジェクトに対して DragOver イベントが発生します。フォームの DragOver イベント プロシージャは、次のような形式になります。

```
SUB Form_DragOver (Source AS CONTROL X AS SINGLE, Y AS SINGLE,
 ⇒ State AS INTEGER)
```

DragOver イベントに対するイベント プロシージャには、DragDrop イベント プロシージャに渡される引数に加えて、*State* 引数が渡されます。*State* 引数の値は、次のような意味を表します。

State の値	意味
0	範囲内に入る。ソースのコントロールは、ターゲットの範囲内に入るところです。
1	範囲外に出る。ソースのコントロールは、ターゲットの範囲外に出るところです。
2	範囲の上。ソースのコントロールは、ターゲット上を移動しています。

一般に、ドラッグされたコントロールが範囲内に入ったときに表示を変え、範囲外に出たときに表示を元に戻すようにします。また、範囲の上で移動して起こるイベントは、マウス ポインタが、オブジェクトの特定の部分 (ピクチャー ボックス内のある領域など) に移動したことを知りたいときに便利です。この場合、引数 *X* と *Y* を使って、ターゲット上でのソースの位置を判断します。

例えば次のプロシージャでは、BackColor および ForeColor プロパティを使って、ユーザーがドラッグしたコントロールが Text1 上に移動したら Text1 の色を反転し、コントロールが Text1 から離れると元の色に戻しています。

```
SUB Text1_DragOver (Source AS CONTROL, X AS SINGLE, Y AS
 ⇒ SINGLE, State AS INTEGER)
    CONST ENTER = 0
    CONST LEAVE = 1
    DIM TempColor AS INTEGER

    IF State = ENTER OR State = LEAVE THEN
        TempColor = Text1.BackColor
        Text1.BackColor = Text1.ForeColor
        Text1.ForeColor = TempColor
    END IF
END SUB
```

この例で使っている定数 ENTER, LEAVE と OVER は、インクルード ファイル CONSTANT.BI で定義されています。

ドラッグの開始と終了の制御

DragMode プロパティを 1 (自動) ではなく 0 (マニュアル) に設定すると、ドラッグ & ドロップの操作をより細かく制御できます。自動のときはコントロールは常にドラッグできるのに対し、マニュアルでは、コントロールがドラッグ可能かどうかを指定することができます。また、マニュアルに設定するとドラッグが開始される前に MouseDown イベントを認識できるので、マウスの位置を記録しておくことが可能です。

フォーム デザイナで新規にコントロールを配置した直後では、DragMode プロパティは 0 (マニュアル) に設定されています。このコントロールをドラッグ可能にするには、コード中で **DRAG** メソッドを使います。**DRAG** メソッドの構文は、次のとおりです。

[*CtlName.*]**DRAG** *action*

action 引数に 1 を指定すると強制的にドラッグが開始され、2 を指定するとドロップされます。また、0 を指定するとドラッグは取り消されます。2 を指定したときは DragDrop イベントが発生しますが、0 を指定したときは発生しません。

例えば、次のステートメントは、ThisBox というコントロールのドラッグを開始します。

```
ThisBox.Drag 1
```

次のステートメントは、コントロールをドロップし、マウス ボタンを離したときと同じ効果をもたらします。

```
ThisBox.Drag 2
```

コントロールの移動

通常は、ユーザーがマウス ボタンを離したら、ソース コントロールの位置を変更します。別のフォームにドラッグできるようにする場合には、この処理は複雑なものになります。あるコントロールを別のフォームに移動したように見せるには、コントロール配列を使って、両方のフォームがそのコントロールを持つように設計しなければなりません。なお、コントロールが一度に 1 つのフォームにだけ表示されるようにする場合は、コントロール配列を使う必要はありません。

1 つのフォーム上で、ドラッグ & ドロップによってコントロールの位置を変えるには、次のように **MOVE** メソッドを使ったイベント プロシージャを記述します。

```
SUB Form_DragDrop (Source AS CONTROL, X AS SINGLE, Y AS SINGLE)
    Source.MOVE X, Y
END SUB
```

ただしこれだけでは、特にサイズの大きなコントロールをドラッグするとき、マウス ポインタが指す位置と実際にドロップされる位置が若干ずれます。ユーザーがコントロールの中の任意の点をクリックできるようにするには、次のようにします。この例では、コントロールの“中心”がマウスの位置になります。

```
SUB Form_DragDrop (Source AS Control, X AS SINGLE, Y AS SINGLE)
    Source.MOVE (X - Source.Width / 2), (Y - Source.Height / 2)
END SUB
```

ソース コントロール内での最初のマウスの位置を記録しておき、コントロールが動かされるときにこの位置をオフセットとして使うと、上の例と同じ操作を行えます。

▼ 最初のマウスの座標を記録してドラッグするには

1. コントロールの DragMode プロパティを 0 (マニュアル) に設定します。
2. x 座標と y 座標をそれぞれ保存するモジュール レベルの変数を宣言します。
3. MouseDown イベントが発生したら、ドラッグを開始します。また、*X* と *Y* の値をモジュール レベルの変数に格納します。
4. MouseUp でドロップします。

次の例は、Picture1 というピクチャー ボックスをドラッグして移動する方法を示しています。このコントロールの DragMode プロパティは、デザイン時に マニュアル (0) に設定しておきます。また、ピクチャー ボックス内での最初のマウス位置を記録しておくモジュール レベルの変数 DragX と DragY を宣言します。

```
' モジュール レベル
DIM SHARED DragX AS SINGLE, DragY AS SINGLE

SUB Picture1_MouseDown (Button AS INTEGER, Shift AS INTEGER, X
 ⇒ AS SINGLE, Y AS SINGLE)
    Picture1.DRAG 1
    DragX = X
    DragY = Y
END SUB

SUB Picture1_MouseUp (Button AS INTEGER, Shift AS INTEGER, X AS
 ⇒ SINGLE, Y AS SINGLE)
    Picture1.DRAG 2
END SUB
```

Form_DragDrop プロシージャでは、実際にコントロールを動かします。ここで、フォームにあるコントロールは Picture1 だけだと仮定すると、ドロップのターゲットになり得るのはフォームだけです。次のように、Form_DragDrop プロシージャは、DragX と DragY をオフセットとして使い、コントロールを再配置します。

```
SUB Form_DragDrop (Source AS Control, X AS SINGLE, Y AS SINGLE)
    Source.MOVE (X - DragX - 1), (Y - DragY - 1)
END SUB
```

コントロール自身へのドロップ

例えば、Picture1 を、それが最初に配置されていた場所の内側でドロップしようとすると、元の位置に戻されてしまいます。これは、Form_DragDrop イベントが発生するのは、フォーム上のコントロールが配置されていない部分で、ソースのコントロールがドロップされたときだけだからです。コントロールの上で離されたときは、それがドラッグされている (ソースの) コントロールが前にあった場所の上であったとしても、そのコントロールの DragDrop イベントだけが発生します。したがって、フォーム内のどこでも Picture1 をドロップできるようにするには、次のようなプログラムを Picture1_MouseDown と Picture1_DragDrop イベントに記述しなければなりません。

```
SUB Picture1_MouseDown (Button AS INTEGER, Shift AS INTEGER, X
 ⇒ AS SINGLE, Y AS SINGLE)
    Picture1.DRAG 1
    DragX = X
    DragY = Y
    StartX = Picture1.Left
    StartY = Picture1.Top
END SUB
```

変数 StartX と StartY は、モジュール レベルで定義します。

```
DIM SHARED StartX AS INTEGER
DIM SHARED StartY AS INTEGER
```

これらの変数は、コントロールの元の位置を記録します。Picture1_DragDrop イベント プロシージャの **MOVE** メソッドで、コントロールの移動量にこれらの値を加え、正しい位置にドロップされるようにします。

```
SUB Picture1_DragDrop (Source AS CONTROL, X AS SINGLE, Y AS
 ⇒ SINGLE)
    Source.MOVE (X - DragX + StartX), (Y - DragY + StartY)
END SUB
```

13.7　フォームが表示されていないときのマウスの利用

Visual Basic for MS-DOS では、フォームが表示されていないときのマウスの使用をサポートするマウス ツールキットが用意されています。

▼ マウス ツールキットを使うには

1. マウス ドライバが正しく設定されていることを確認します。
2. /L オプションでデフォルトの Quick ライブラリを指定して、プログラミング環境を起動します。

   ```
   VBDOS /L VBDOS.QLB
   ```

3. [ﾌｧｲﾙ] メニューから [ファイル追加] コマンドを選択して MOUSE.BAS を選択します。

マウス ツールキットには、次のようなプロシージャが含まれています。

MouseInit	マウス ドライバを初期化する。
MousePoll	マウスの位置とマウス ボタンのステータスを返す。
MouseShow	マウス ポインタを表示する。
MouseHide	マウス ポインタを消す。
MouseBorder	マウス ポインタの移動を制限する境界を設定する。
MouseDriver	マウスが存在するかどうかをチェックし、マウス ドライバ関数へのアクセスを可能にする。

重　要　フォームが表示されているときには、マウス ツールキットのプロシージャを使わないでください。Visual Basic が標準で用意しているマウス イベントと競合する可能性があります。

各プロシージャの使い方や引数については、それぞれのコメントを参照してください。

14 アプリケーションの外観の変更

アプリケーションの作成が一通り完了したら、次に **SCREEN** オブジェクトの ControlPanel プロパティを使って、画面各部の背景色や文字色など、アプリケーションの外観を整えます。

14.1 ControlPanel プロパティの使い方

ControlPanel プロパティは、**SCREEN** オブジェクトの持つさまざまな機能を設定します。外観の変更の一部は、各コントロールやフォームごとに BackColor プロパティなどを使って行うことができますが、**SCREEN** オブジェクトの ControlPanel プロパティを使うと、画面全体に対して、より詳細な設定が可能です、この設定は、Microsoft Windows のコントロール パネルで行う設定に似ています。

ControlPanel プロパティは他のプロパティと異なり、18 個の要素からなる整数配列になっています。各要素には、配列の添字に相当する 0 ～ 17 の番号が割り当てられていて、それぞれ別の画面属性に対応しています。ControlPanel 配列で設定できる画面属性とそのデフォルトの設定は次のとおりです。

表 14.1　ControlPanel プロパティ

画面属性	要素を示す値	デフォルト値
アクセス キーの文字色	(0)	15 (明るい白)
アクティブなウィンドウの境界の背景色	(1)	7 (白)
アクティブなウィンドウの境界の文字 (前景) 色	(2)	0 (黒)
アクティブなウィンドウの影の有無	(3)	-1 (True)
コマンド ボタンの文字色	(4)	0 (黒)
デスクトップの背景色	(5)	0 (黒)
デスクトップの文字色	(6)	7 (白)
デスクトップの背景の文字パターン	(7)	32 (スペース)
無効に設定されている項目の文字色	(8)	8 (灰)
メニューの背景色	(9)	7 (白)
メニューの文字色	(10)	0 (黒)
選択 (反転表示) されているメニュー項目の背景色	(11)	0 (黒)
選択 (反転表示) されているメニュー項目の文字色	(12)	7 (白)
フォーム上に独立して配置されているスクロール バーの背景色	(13)	7 (白)

フォーム上に独立して配置されているスクロール バーの文字色	(14)	0 (黒)
3-D 表示の有無	(15)	-1 (True)
アクティブなフォームのタイトル バーの背景色	(16)	0 (黒)
アクティブなフォームのタイトル バーの文字色	(17)	15 (明るい白)

色の設定を行う要素に対しては、0 ～ 15 の色番号が指定できます。また、要素 3 と 15 に対しては、True (-1) または False (0) を指定できます。

デザイン時には、ControlPanel プロパティに値を設定することはできません。フォーム デザイナの [補助] メニューの [画面設定] コマンドを使うと、デザイン時の表示色が設定できます。ただしこの設定が有効なのはフォーム デザイナ上だけで、アプリケーションの実行時の設定は、デフォルトで上の表のとおりになりますので注意してください。

例えば、[補助] メニューの [画面設定] コマンドのダイアログ ボックスでは、フォームのタイトル バーの色を変更できますが、そのフォームが含まれるアプリケーションを実行すると、タイトル バーの色はデフォルト値に戻ってしまいます。コード中から ControlPanel プロパティの値を変更すれば、実行時に設定どおりのタイトル バーの色を表示できます。

ControlPanel 配列への値の代入

ControlPanel プロパティを設定するには、他のプロパティに対する設定と同様に代入文を使いますが、配列の添字を指定しなければならない点が異なります。ControlPanel 配列の要素への値の代入は、次の構文で行います。

SCREEN.ControlPanel(*arrayitem%*) = *value%*

引数	説明
arrayitem%	配列の要素を示す 0 ～ 17 の値 (添字)。定数や変数で指定してもかまいませんし、数式で指定することもできます 。また、インクルード ファイル CONSTANT.BI で定義されている記号定数を使うこともできます。
value%	各要素の設定を示す整数。

arrayitem% には、整数値を指定する代わりに、インクルード ファイル CONSTANT.BI で定義されている記号定数を使うこともできます。次の定数が定義されています。

CONSTANT.BI で定義されている定数	配列の要素を示す値 (添字)
ACCESSKEY_FORECOLOR	0
ACTIVE_BORDER_BACKCOLOR	1
ACTIVE_BORDER_FORECOLOR	2
ACTIVE_WINDOW_SHADOW	3
COMBUTTON_FORECOLOR	4
DESKTOP_BACKCOLOR	5
DESKTOP_FORECOLOR	6

DESKTOP_PATTERN	7
DISABLED_ITEM_FORECOLOR	8
MENU_BACKCOLOR	9
MENU_FORECOLOR	10
MENU_SELECTED_BACKCOLOR	11
MENU_SELECTED_FORECOLOR	12
SCROLLBAR_BACKCOLOR	13
SCROLLBAR_FORECOLOR	14
THREE_D	15
TITLEBAR_BACKCOLOR	16
TITLEBAR_FORECOLOR	17

▼ CONSTANT.BI で定義されている記号定数を使うには

- CONSTANT.BI を別のファイル名でコピーして、必要な定数だけを残して削除し、定数を利用するすべてのモジュールに **$INCLUDE** メタコマンドを記述してインクルードします。または CONSTANT.BI から、必要な定数の宣言を、定数を利用するすべてのモジュールのモジュール レベルにコピーします。

定数の宣言は、その定数が使われるよりも前に行わなければなりません。インクルード ファイルをインクルードするには、モジュール レベルに **$INCLUDE** メタコマンドを記述します。例えば、CNTRL.BI をインクルードするには次のようにします。

```
'$INCLUDE: 'CNTRL.BI'
```

注 意 インクルード ファイル CONSTANT.BI には、さまざまな定数が定義されています。このファイル全体をアプリケーションにインクルードすると、実行時に余分なメモリを消費してしまいます。そのアプリケーションで使う定数だけをコピーして使うことをおすすめします。

文字色と背景色の設定

文字色および背景色として有効な色番号は、次のとおりです。それぞれの色を示す記号定数が、インクルード ファイル CONSTANT.BI で定義されています。

色番号	表示色	CONSTANT.BI
0	黒	BLACK
1	青	BLUE
2	緑	GREEN
3	水	CYAN
4	赤	RED
5	紫	MAGENTA
6	黄	BROWN
7	白	WHITE

8	灰	GRAY
9	青(明)	BRIGHT_BLUE
10	緑(明)	BRIGHT_GREEN
11	水(明)	BRIGHT_CYAN
12	赤(明)	BRIGHT_RED
13	紫(明)	PINK
14	黄(明)	YELLOW
15	白(明)	BRIGHT_WHITE

次の例は、アクティブなコマンド ボタンの文字色を緑色に設定しています。

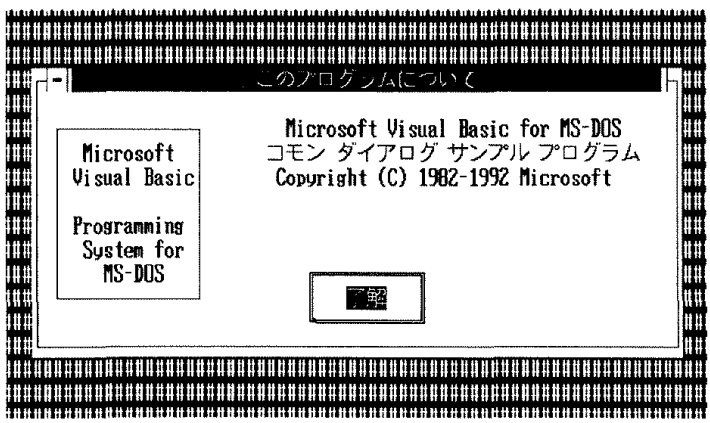
```
SCREEN.ControlPanel(4) = 2
```

記号定数を使うと、次のように記述できます。

```
CONST COMBUTTON_FORECOLOR = 4
CONST GREEN = 2
SCREEN.ControlPanel(COMBUTTON_FORECOLOR) = GREEN
```

デスクトップ パターンの設定

デスクトップとは、フォームの背景となる画面全体を指します。



デスクトップは、1文字のASCII文字で埋めることができます。デフォルトでは、このキャラクタはスペース(ASCIIコード32)に設定されているため、何も表示されていないように見えます。デスクトップの背景の文字パターンを設定するのはControlPanel(7)です。

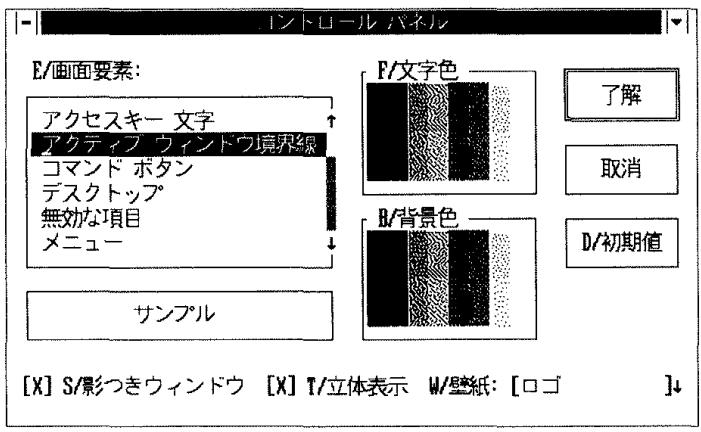
ヒント　ControlPanel(7)を0に設定すると、デスクトップの背景の文字パターンはnull文字("")に設定されます。この機能を使って、独自のデスクトップ パターン(壁紙)を作成することができます。第11章「デスクトップ出力とキーボード入力」の例を参照してください。

影と 3D 機能

影とはウィンドウを立体的に見せるもので、そのときアクティブなウィンドウの右と下に表示されます。また、Visual Basic の各コントロールは、3D 機能で立体的に表示されます。例えば、コマンド ボタンは手前に飛び出しているように表示され、クリックするとへこむように見えます。デフォルトでは、両方とも True (-1) に設定されています。

14.2 ControlPanel サンプル アプリケーション

Visual Basic に付属している ControlPanel サンプル アプリケーションを利用すると、実行時にダイアログ ボックスを表示して ControlPanel 配列の値を設定できるようになります。ControlPanel サンプル アプリケーションの機能を見るには、プログラミング環境で CONTROLP.MAK を読み込み、実行してみてください。



ユーザーはダイアログ ボックスから、設定を変更したい画面の部分を選択します。[文字色] ボックス，[背景色] ボックス，[壁紙] コンボ ボックスで色やデスクトップ パターンの設定を行い、ControlPanel 配列の要素を設定できます。また、チェック ボックスでは、3-D 機能とウィンドウの影のオン / オフを切り替えることができます。

ControlPanel アプリケーションは、初期化ファイル (CONTROLP.INI) を使って、ユーザーが設定した内容を保存するようになっています。

▼ **開発中のプロジェクトに ControlPanel サンプル アプリケーションを追加するには**

1. [ﾌｧｲﾙ] メニューから [ファイル追加] コマンドを選択し、[ファイル] リスト ボックスから CONTROLP.FRM ファイルを選択してロードします。
2. 同じ手順で、CONTDRAW.BAS をロードします。
3. コントロール メニュー ボックスと 最小化ボタンをダイアログ ボックスから取り除きたいときは、ControlBox プロパティと MinButton プロパティを False (0) に設定します。

4. ダイアログ ボックスをロードして表示するために、次のステートメントをコード中の適切な位置に追加します。

```
frmControlPanel.SHOW
```

フォームをモーダル ダイアログ ボックスとして表示するときは、**SHOW** メソッドの引数に値 1 を指定してください。例えば、このステートメントを [画面表示設定] というメニュー コマンドの Click イベント プロシージャに記述した場合、実行時にユーザーが [画面表示設定] コマンドをクリックすると、ControlPanel ダイアログ ボックスが表示されます。

注　意　アプリケーションの起動時に画面属性を.INI ファイルの設定値で初期化したいときは、Form_Load イベント プロシージャで ControlPanel フォームをロードします。ただし、ユーザーが ControlPanel ダイアログ ボックスをオープンするまでは表示しないように設定してください。

---

15 プログラムの流れの制御

条件判断やループといった制御構造を使うと、プログラムの実行の流れをコントロールできます。条件判断ステートメントは、複数の選択肢から 1 つを選び実行します。ループは、プログラムの一連のステートメントを必要な回数だけ繰り返し実行します。

15.1 ステートメントの実行順序の変更

Basic では原則として、制御構造を持たないプログラムのステートメントは、左から右に、上から下に向かって実行されます。このような単純な流れだけでも簡単なプログラムを書くことができますが、条件判断ステートメントとループを使ってステートメントの実行順序を変更すると、プログラミング言語の機能をより柔軟に利用できます。

フォームを使わないアプリケーションでは、プログラムの実行の流れは原則のとおりです。フォームを使うアプリケーションでは、イベントの発生が、プログラムの流れを決めます。イベントが発生すると、Visual Basic がどのイベント プロシージャを実行するのかを判断して、適切な位置に自動的にプログラムの制御を移します。

条件判断を使うと、式を評価した結果に従って、関連するステートメント (ステートメント ブロック) の 1 つに分岐させることができます。例えば、コントロール配列に含まれるコマンド ボタンの 1 つをユーザーがクリックしたとき、引数としてイベント プロシージャに渡されたインデックス番号を基に、どのコマンド ボタンがクリックされたかを判断して、実行する処理を決めることができます。また、ループを使うと、一連のステートメントを繰り返し実行できます。

15.2 論理式

論理式とは、真 (-1) または偽 (0) の値を返す式のことです。ブール式 (boolean expression) と呼ぶこともあります。ほとんどの条件判断とループでは論理式を使います。次の **IF...THEN...ELSE** ステートメントには、論理式 X < Y が含まれています。

```
IF X < Y THEN CALL Procedure1 ELSE CALL Procedure2
```

この例では、論理式が真のとき (変数 X の値が変数 Y の値よりも小さいとき)、Procedure1 が実行され、偽のとき (X が Y 以上のとき)、Procedure2 が実行されます。

この例が示すように、論理式は一般に、異なる 2 つの式 (この場合は X と Y) を比較して、それらの関係を判定するために使います。比較には、次の表に示す関係演算子を使います。

表 15.1　Visual Basic で使う関係演算子

演算子	意味
=	等しい
<>	等しくない
<	より小さい
<=	以下
>	より大きい
>=	以上

これらの関係演算子は、数式と文字列式の両方の比較に使うことができます。文字列式の比較では、より大きい，より小さいなどの関係は、文字コード順によって決まります。例えば、次の式の deduce という単語は deduct という単語よりも文字コード順で前に来る (小さいと判断される) ため、結果は真になります。

```
"deduce" < "deduct"
```

文字コードの順序は ASCII コードおよび漢字連続コードに従います。漢字連続コードは JIS 漢字コードおよびシフト JIS コードの空きコードを詰めたものであり、各文字の順序はこれらのコードと同じです。1 バイトコードは 2 バイトコードより小さいと判断されます。

また論理式には、2 つの式の関係を論理的に判断して真または偽の結果を返す、論理演算子 **AND, OR, NOT, XOR, IMP, EQV** を使うことができます。論理演算子を使うと、複数の式を組み合わせて、複雑なテストを行うことができます。論理演算子とその機能を次の表に示します。

演算子	真を返すとき	この演算子を使う場合
AND	どちらのオペランドも真のとき	2 つの式がともに真であることを確認するとき
OR	一方のまたは両方のオペランドが真のとき	少なくとも 1 つの式が真であることを確認するとき
XOR	どちらか一方のオペランドだけが真のとき	2 つの式のうち一方だけが真であることを確認するとき
NOT	オペランドが偽のとき	ビットを反転させたいとき
IMP	最初のオペランドが偽、または 2 番目のオペランドが真のとき	条件式を短く表現したいとき
EQV	両方のオペランドが真か、両方のオペランドが偽のとき	2 つの式が両方とも真または偽であることを確認するとき

例えば次の式は、expression1 と expression2 が両方とも真のときだけ、結果が真になります。

```
expression1 AND expression2
```

したがって、次の例では、論理式 X <= Y と Y <= Z が両方とも真のときだけ、“ソート終了” というメッセージが表示されます。

```
IF (X <= Y) AND (Y <= Z) THEN MSGBOX ("ソート終了")
```

注 意 <= などの関係演算子は **AND** などの論理演算子よりも先に評価されます。したがって上の例では、2 つの論理式をカッコで囲まなくても結果は同じです。しかしカッコを使うと、複雑な論理式が読みやすくなるだけでなく、式の評価順序を確実に指定できます。

Visual Basic では、-1 が真 (True) を表し、0 が偽 (False) を表します。次の例のように真の式と偽の式を表示すると、これら値を確認することができます。

```
X = 5
Y = 10
MSGBOX STR$(X < Y)        '論理式を評価して，「真」の値を出力します．
MSGBOX STR$(X > Y)        '論理式を評価して，「偽」の値を出力します．
```

上の例を実行すると、ダイアログ ボックスには次のように表示されます。

```
-1
 0
```

論理式で、偽を表す値は必ず 0 です。また、Visual Basic が論理式を評価して結果が真であるときは、値として必ず -1 を返します。ただし論理式の中で使う数式の値は、0 以外はすべて真と見なされます。次の例を実行すると確認できます。

```
X = VAL(INPUTBOX$ ("値を入力してください:"))
IF X THEN MSGBOX STR$(X) + " は真です."
```

最初の行で、ダイアログ ボックスに次のプロンプトが表示されます。ここでは 2 と入力してみます。

値を入力してください:**2**

2 行目で結果が評価され、ダイアログ ボックスには次のように表示されます。

2 は真です.

Visual Basic の **NOT** 演算子は、ビット単位の演算を行う演算子です。C や Pascal など、プログラミング言語の中には、ビット単位の演算を行う **NOT** 演算子と論理演算を行う **NOT** 演算子の両方を持つものがあります。2 つの演算子の違いは次のとおりです。

- ビット単位の演算を行う **NOT** 演算子は、値 -1 に対してだけ偽 (0) を返します。
- 論理演算を行う **NOT** 演算子は、0 以外のすべての値に対して偽 (0) を返します。

次の表に示すように、ビット単位の演算では、*expression* の値が -1 のときだけ偽 (0)、*expression* の値が -1 以外のときはすべて真 (0 以外) になります。

expression の値	NOT *expression* の値
1	-2
0	-1
-1	0
-2	1

NOT *expression* が偽になるのは、*expression* を評価した値が -1 になったときだけであることに注意してください。このため、プログラムで論理値を持った定数や変数を定義するときには、真を表す値として必ず -1 を使います。また、Visual Basic で論理演算を行う **NOT** 演算子をシミュレートするには、関係演算子 <> を使います。

Visual Basic の **NOT** 演算子は、オペランドの値を 2 進数で表したとき、各ビットを反転します。1 のビットは 0 に、0 のビットは 1 になります。整数値 0 (偽) は、内部的には 16 個の 0 のビットの並びで表されるので、NOT 0 (真) は、内部的には 16 個の 1 のビットの並びで表されることになります。

```
FALSE = 0000000000000000
TRUE = NOT FALSE = 1111111111111111
```

Visual Basic では整数値を 2 の補数で扱うので、1 のビットの 16 個の並びは、-1 という値になります。

論理値の 0 と -1 を記号定数として定義すると、ループや条件判断で使うときに便利です。次の例では、この方法が使われています。このプログラムは、Amount という名前の配列の要素を昇順にソートします。

```
'プログラムで使う記号定数を定義します.
CONST FALSE = 0, TRUE = NOT FALSE
   .
   .
   .
DO
    Swaps% = FALSE
    FOR I% = 1 TO TransacNum - 1
        IF Amount(I%) < Amount(I%+1) THEN
            SWAP Amount(I%), Amount(I%+1)
            Swaps% = TRUE
        END IF
    NEXT I%
LOOP WHILE Swaps%              'Swaps% が真 (TRUE) になるまで
                               'ループを繰り返します.
```

記号定数の TRUE と FALSE は、標準で用意されているインクルード ファイル CONSTANT.BI で定義されています。このファイルには他にもいくつかの定数が定義されており、プログラムを読みやすくするために便利です。

15.3 条件判断

プログラムの中で条件判断を使うと、式の値によって、次のどちらかの動作が行われます。

- 条件判断の構造の中に記述されているいくつかのステートメントの集まり (ステートメント ブロック) のうちの 1 つを実行します。
- 条件判断の構造を抜け、プログラムの別の部分に分岐します。

Visual Basic では、条件判断の構造として、1 行形式およびブロック形式の **IF...THEN** [...**ELSE**] ステートメントと、**SELECT CASE...END SELECT** ステートメントを使うことができます。

1 行形式の IF...THEN...ELSE

簡単な条件判断は、1 行形式の **IF...THEN** [...**ELSE**] ステートメントで記述できます。最も簡単な形式 (**IF...THEN**) では、**IF** キーワードに続く式が評価されます。式の値が真であれば、**THEN** キーワードに続くステートメントが実行されます。式の値が偽であれば、**IF...THEN** ステートメントの次の行から処理が続けられます。

IF...THEN [...**ELSE**] ステートメントの構文を次に示します。

IF *condition1* **THEN** *statementblock-1* [**ELSE** *statementblock-2*]

次の例では、変数 A の値と変数 B の値が等しい（条件式の値が真）とき、メッセージが表示されます。

```
IF A = B THEN MSGBOX "2 つの値は等しい"
```

同じ行に **ELSE** キーワードを付けると、式の値に応じて処理を分岐できます。式の値が真であれば、**THEN** キーワードの後ろの処理が実行されます。式の値が偽であれば、**ELSE** キーワードの後ろの処理が実行されます。

```
IF A <> B THEN MSGBOX "等しくない" ELSE MSGBOX "等しい"
```

THEN および **ELSE** キーワードの後に複数のステートメントを指定したいときは、コロン (:) で区切って続けます。

IF...THEN [...**ELSE**] ステートメントはネストする (**IF...THEN** [...**ELSE**] の中に、さらに別の **IF...THEN** [...**ELSE**] を入れる）ことができます。ただし、1 行に多くのステートメントを記述するとプログラムが読みにくくなるだけでなく、デバッグも難しくなります。複雑な条件判断を実行したいときは、ブロック形式の **IF...THEN...ELSE** ステートメントや **SELECT CASE** ステートメントを使います。

ブロック形式の IF...THEN...ELSE

表 15.2 に、ブロック形式の **IF...THEN...ELSE** ステートメントの構文と使用例を示します。

表 15.2　ブロック形式の IF...THEN...ELSE ステートメントの構文と使用例

構文	使用例

IF *condition1* **THEN**
　[*statementblock-1*]
[**ELSEIF** *condition2* **THEN**
　[*statementblock-2*]]...
[**ELSE**
　[*statementblock-n*]]
ENDIF
	`IF X > 0 THEN`
	`　MSGBOX "X は正"`
	`　PosNum = PosNum + 1`
	`ELSEIF X < 0 THEN`
	`　MSGBOX "X は負"`
	`　NegNum = NegNum + 1`
	`ELSE`
	`　MSGBOX "X は 0"`
	`END IF`

condition1 や *condition2* などの引数には、数式または論理式を指定します。数式の場合は 0 以外の値が真になり、0 が偽になります。また、論理式の場合は、-1 が真になり、0 が偽になります。論理式は一般に、<, > , = などの関係演算子を使って、2 つの数式や文字列式を比較したものです。

IF, **ELSEIF**, **ELSE** のそれぞれに続く行には、いくつかのステートメントの集まりであるステートメント ブロックを置きます。このとき、ブロック内のステートメントを **IF**, **ELSEIF**, **ELSE** キーワードと同じ行に書くことはできません。同じ行にステートメントを書くと、Visual Basic は 1 行形式の **IF...THEN** ステートメントとして扱います。

Visual Basic は、**IF** と **ELSEIF** の行の式を上から順に評価し、最初の真の式が見つかるまでステートメント ブロックをスキップします。真の式が見つかると、続いて記述されているステートメント ブロックを実行してから、**END IF** の次の行に分岐します。

IF や **ELSEIF** の行の式がすべて偽のときは、**ELSE** キーワード があればそれに続くステートメント ブロックが実行されます。**ELSE** キーワードがないときは、**END IF** の次の行から実行を続けます。偽のときに実行すべきステートメントがなければ、**ELSE** や **ELSEIF** は省略できます。

ブロック形式の **IF...THEN...ELSE** ステートメントでは、次の例のように複数の **ELSEIF** を含めることもできます。

```
IF C$ >= "A" AND C$ <= "Z" THEN
    MSGBOX "大文字です"
ELSEIF C$ >= "a" AND C$ <= "z" THEN
    MSGBOX "小文字です"
ELSEIF C$ >= "0" AND C$ <= "9" THEN
    MSGBOX "数字です"
ELSE
    MSGBOX "英数字ではありません"
END IF
```

ただし、複数の条件が真になるときでも、実行されるステートメント ブロックは(最初に見つかった真の式に続く)1つだけです。例えば、次のプログラムで、データとしてaceという単語を入力すると、メッセージ“入力が短すぎます” が表示されますが、メッセージ“a から始めてはいけません” は表示されません。

```
Check$ = INPUTBOX$ ("単語を入力してください:")
IF LEN(Check$) > 6 THEN
    MSGBOX "入力が長すぎます"
ELSEIF LEN(Check$) < 6 THEN
    MSGBOX "入力が短すぎます"
ELSEIF LEFT$(Check$, 1) = "a" THEN
    MSGBOX "a から始めてはいけません"
END IF
```

IF...THEN...ELSE ステートメントは、ネストすることができます。次の例のように、ある **IF...THEN...ELSE** ブロックの中に別の **IF...THEN...ELSE** ブロックを入れることができます。

```
IF X > 0 THEN
    IF Y > 0 THEN
        IF Z > 0 THEN
            MSGBOX "すべて 0 より大きい値です."
        ELSE
            MSGBOX "X と Y だけが 0 より大きい値です."
        END IF
    END IF
ELSEIF X = 0 THEN
    IF Y = 0 THEN
        IF Z = 0 THEN
            MSGBOX "すべて 0 です."
        ELSE
            MSGBOX "X と Y だけが 0 です."
        END IF
    END IF
ELSE
    MSGBOX "X は 0 より小さい値です."
END IF
```

SELECT CASE ステートメント

SELECT CASE ステートメントは、ブロック形式の**IF...THEN...ELSE** ステートメントと同じように、複数の選択肢からなる条件判断を行います。これら2つのステートメントの大きな違いは、**SELECT CASE** で条件判断のために評価できる式は1つの式だけであるのに対し、**IF...THEN...ELSE** はまったく別々の式を評価できる点です。ブロック形式の**IF...THEN...ELSE** ステートメントは、**SELECT CASE** ステートメントの代わりに使うことができます。ただし、1つの式を評価した結果で複数の処理に分岐させたいときには、**SELECT CASE** ステートメントを使った方が、読みやすく理解しやすいプログラムになります。

次の例は、**SELECT CASE** ステートメントと **IF...THEN...ELSE** ステートメントの類似点と相違点を示したものです。まず最初に、ブロック形式の **IF...THEN...ELSE** ステートメントを使った、複数の選択肢による条件判断を示します。

```
X = VAL(INPUTBOX$ ("1～3 の整数を入力してください"))
IF X = 1 THEN
    MSGBOX "1"
ELSEIF X = 2 THEN
    MSGBOX "2"
ELSEIF X = 3 THEN
    MSGBOX "3"
ELSE
    MSGBOX "1～3 の整数を入力してください"
END IF
```

上の条件判断は、**SELECT CASE** を使って次のように書き直すことができます。

```
X = VAL(INPUTBOX$ ("1～3 の整数を入力してください"))
SELECT CASE X
    CASE 1
        MSGBOX "1"
    CASE 2
        MSGBOX "2"
    CASE 3
        MSGBOX "3"
    CASE ELSE
        MSGBOX "1～3 の整数を入力してください"
END SELECT
```

次の条件判断では、複数の条件式が必要になるため、**SELECT CASE** ステートメントを使うことはできず、**IF...THEN...ELSE** ステートメントを使います。

```
X = VAL(INPUTBOX$("最初の値を入力してください"))
Y = VAL(INPUTBOX$("2 番目の値を入力してください"))
IF X = 0 AND Y = 0 THEN
    MSGBOX "どちらも 0 です."
ELSEIF X = 0 THEN
    MSGBOX "X だけが 0 です."
ELSEIF Y = 0 THEN
    MSGBOX "Y だけが 0 です."
ELSE
    MSGBOX "どちらも 0 ではありません."
END IF
```

SELECT CASE ステートメントの使い方

表 15.3 に、**SELECT CASE** ステートメントの構文と使用例を示します。

表 15.3 SELECT CASE ステートメントの構文と使用例

構文	使用例

SELECT CASE *testexpression*
CASE *expressionlist1*
[*statementblock-1*]
[**CASE**] *expressionlist2*
[*statementblock-2*]]...
.
.
.
[**CASE ELSE**
[*statementblock-n*]]
END SELECT
	`TstVal = VAL(INPUTBOX$("数値を入力してください")`
	`SELECT CASE TstVal`
	`CASE 1, 3, 5, 7, 9`
	`MSGBOX "奇数"`
	`CASE 2, 4, 6, 8`
	`MSGBOX "偶数"`
	`CASE IS < 1`
	`MSGBOX "小さすぎます"`
	`CASE IS > 9`
	`MSGBOX "大きすぎます"`
	`CASE ELSE`
	`MSGBOX "整数ではありません"`
	`END SELECT`

CASE キーワードに続く引数 *expressionlist* は、次のいずれかの式か、それらをカンマで区切って並べたものです。

- 数式，または特定の範囲を表す数式
- 文字列式，または特定の範囲を表す文字列式

特定の範囲を表す式を指定するときは、次のような構文を使います。

CASE *expression* **TO** *expression*
CASE IS *relational-operator expression*

relational -operator には、表 15.1 に示す関係演算子を指定します。**TO** キーワードを使って範囲を表すときは、小さい値を先に指定してください。大きい値を先に指定すると、対応するステートメントは実行されません。例えば、-5 から -1 までの範囲を指定するときは、次のようにします。

```
CASE -5 TO -1
```

同様に、特定の範囲を表す文字列式では、次の例のように文字コード順で前に来る文字列を先に指定します。

```
CASE "aardvark" TO "bear"
```

それぞれの **CASE** キーワードの後に、複数の式または範囲を表す式を並べて指定できます。**IS** キーワードは、プログラムを読みやすくするために入れるもので、省略することもできます。

```
CASE 1 TO 4, 7 TO 9, WildCard1%, WildCard2%
CASE IS = Test$, IS = "データ終了"
CASE IS < LowerBound, 5, 6, 12, IS > UpperBound
CASE IS < "HAN", "MAO" TO "TAO"
```

SELECT CASE ステートメントで指定された値（次の例では **X**）が、**CASE** キーワードに続く式のリスト中に現れるときは、その **CASE** の行に続くステートメント ブロックが実行されます。実行が終了すると、**END SELECT** の次の行から実行が続きます。

```
X = VAL(INPUTBOX$("1, 2 または 3 を入力してください"))
SELECT CASE X
    CASE 1
        Msg$ = "1, "
    CASE 2
        Msg$ = "2, "
    CASE 3
        Msg$ = "3, "
END SELECT
MSGBOX Msg$ + "これで終わりです"
```

この例の1行目が実行されると、ダイアログ ボックスに次のプロンプトが表示され、ユーザーの入力待ちになります。

1, 2 または 3 を入力してください

1 を入力して [ENTER] キーを押すと、次のメッセージが表示されます。

1, これで終わりです

同じ値または範囲を複数の **CASE** キーワードの後に指定しても、式は上から順に評価されるため、実際に実行されるのは最初に一致したものだけです。

```
Test$ = INPUTBOX$("ABORIGINE と入力してください")
SELECT CASE Test$
    CASE "A" TO "AZZZZZZZZZZZZZZZZZ"
        MSGBOX "A から始まる大文字の単語です"
    CASE IS < "A"
        MSGBOX "アルファベット以外の文字の列"
    CASE "ABORIGINE"
            'ABORIGINE は、最初の CASE の後に指定した範囲に含まれ
            'るため、この CASE が実行されることはありません。
        MSGBOX "特殊な場合です"
END SELECT
```

この例の1行目を実行すると、ダイアログ ボックスに次のプロンプトが表示されます。

ABORIGINE と入力してください

ABORIGINE と入力すると、次のメッセージが表示されます。

A から始まる大文字の単語です

CASE ELSE は、すべての **CASE** の後に記述します。**CASE ELSE** と **END SELECT** の間のステートメントは、**SELECT CASE** の引数 *testexpression* で指定した値が、**CASE** キーワードの後に指定されたどの条件とも一致しない場合に実行されます。*testexpression* の値が予測できないときに、**CASE ELSE** を指定するようにします。

CASE キーワードの後に指定されたどの条件とも一致せず、**CASE ELSE** も指定されていない場合は、**SELECT CASE ... END SELECT** ブロック中のステートメント ブロックは 1 つも実行されません。

次の例の ButtonArray は、コマンド ボタンのコントロール配列です。**SELECT CASE** ステートメントは、ユーザーがクリックしたコマンド ボタンのインデックス番号 (index プロパティの値) に対応したプロシージャを実行します。

```
SUB ButtonArray_Click(Index AS INTEGER)
    'SELECT CASE を使って，ユーザーの選択に対応した処理を行います．
SELECT CASE Index
    CASE 1
        CALL AddData
    CASE 2
        CALL DeleteData
    CASE 3
        CALL ChangeData
    CASE 4
        CALL ListData
  .
  .
  .
```

15.4 ループ構造

ループ構造とは、指定した回数だけ、または式（ループ条件）が真または偽になるまで、ステートメント ブロックを繰り返し実行するものです。Visual Basic には、次のループ構造が用意されています。

- **FOR...NEXT** ステートメント
- **WHILE...WEND** ステートメント
- **DO...LOOP** ステートメント

FOR...NEXT ループ

FOR...NEXT ループは、ループ内のステートメントを指定した回数だけ実行します。この回数は、ループ カウンタを開始値から終了値まで増減させてカウントし、ループ カウンタが指定した終了値に達するまでループの実行が続れられます。次の表は、**FOR...NEXT** ステートメントの構文と使用例を示しています。

表 15.4 FOR...NEXT ステートメントの構文と使用例

構文	使用例

FOR *counter* = *start* **TO** *end* [**STEP** *increment*]
[*statementblock-1*]
[**EXIT FOR**]
[*statementblock-2*]
NEXT [*counter*] [, *counter*]...
	`FOR I% = 1 TO 10`
	`  Array%(I%) = I%`
	`NEXT I%`

FOR...NEXT ループを実行するとき、ループ カウンタとして使われる変数 *counter* には、最初に式 *start* の値が設定されます。ループを繰り返すたびに *counter* に値が加算されます。オプションの **STEP** キーワードを省略すると、加算される値のデフォルト値は 1 になり、ループが実行されるたびに *counter* に 1 が加算されます。**STEP** キーワードを使うと、引数 *increment* で指定した値が *counter* に加算されます。引数 *increment* には任意の数値を指定できます。負の値を指定すると、*start* から *end* までカウント ダウンしながらループが実行されます。次のどちらかの条件が真であれば、ループは終了します。

- *increment* が正のとき、*counter* をカウント アップして、その値が *end* よりも大きくなったとき
- *increment* が負のとき、*counter* をカウント ダウンして、その値が *end* よりも小さくなったとき

FOR...NEXT ステートメントは、つねにループの先頭で条件をテストします。したがって次のいずれかの条件が真のときは、ループは 1 度も実行されません。

- *increment* が正の値で、*start* の初期値が *end* よりも大きいとき

```
'I% の初期値が 9 よりも大きいので、ループは実行されません.
FOR I% = 10 TO 9
   .
   .
   .
NEXT I%
```

- *increment* が負の値で、*start* の初期値が *end* よりも小さいとき

```
'I% の初期値が -9 よりも小さいので、ループは実行されません.
FOR I% = -10 TO -9 STEP -1
   .
   .
   .
NEXT I%
```

NEXT の引数 *counter* は省略可能です。ただし、**FOR...NEXT** ループをネストするときは、*counter* を指定しておくと、ループの構造がわかりやすくなります。

次に、**FOR...NEXT** ループをネストする場合の一般的なガイドラインを示します。

- **NEXT** にループ カウンタ変数を指定するときは、ネストしたループのカウンタは、外側のループのカウンタよりも前に指定します。次の例は、ループを正しくネストしています。

```
FOR I = 1 TO 10
    FOR J = -5 TO 0
   .
   .
   .
    NEXT J
NEXT I
```

次のようなループのネストは正しくありません。このようなプログラムを実行すると、"FOR がありません" というエラー メッセージが表示されます。

```
FOR I = 1 TO 10
    FOR J = -5 TO 0
  .
  .
  .
    NEXT I        'ここでは NEXT J とする必要があります.
NEXT J            'ここでは NEXT I とする必要があります.
```

- ループ カウンタ変数に整数を使うと、生成されるコードが小さくなり、ループの実行速度が上がります。また、プログラミング環境で実行するときは、**NEXT** に対して *counter* を省略すると、ループの実行速度が上がります。カウンタの指定を省略するときは、プログラムがわかりにくくならないよう、コメントに書くようにしてください。

```
FOR I = 1 TO 20
  .
  .
  .
NEXT          'カウンタは I
```

  実行可能 (.EXE) ファイルとして実行する場合は、**NEXT** に対してループ カウンタを指定するかどうかは、実行速度に影響しません。

- **FOR** と **NEXT** の数は、同じ数でなければなりません。

- 1 つの **NEXT** キーワードに対して複数のループ カウンタを指定し、複数の **FOR...NEXT** ループを終了させるときは、内側のループのループカウンタから順に並べなければなりません。

  NEXT (最も内側のループ カウンタ), ... , (最も外側のループ カウンタ)

  このとき、**NEXT** で指定するループカウンタの個数は、**FOR** の個数と一致していなければなりません。

- **FOR** に続く条件式では、*start%* と *end%* はループ開始の時点でだけ評価されます。

```
FOR I% = GetStart% TO n%+2
  .
  .
  .
NEXT I%
```

この例では、ループを実行する間に GetStart% や n% の値が変化しても、ループの実行回数には影響しません。ただし、カウンタの値 (この場合は I%) を変更するとループの実行回数に影響します。このようなプログラミングは避けることをおすすめします。

次の 3 つのプログラムは、それぞれ別々の方法で **FOR...NEXT** ループをネストしていますが、いずれも同じ結果を出力します。最初の例では、ループごとに別々の **NEXT** とループ カウンタを使っています。

```
FOR I = 1 TO 2
    FOR J = 4 TO 5
        FOR K = 7 TO 8
            PRINT I, J, K
        NEXT K
    NEXT J
NEXT I
```

次の例でもループ カウンタを使っていますが、1つの **NEXT** ですべてのループを終了しています。

```
FOR I = 1 TO 2
    FOR J = 4 TO 5
        FOR K = 7 TO 8
            PRINT I, J, K
NEXT K, J, I
```

最後は、**NEXT** でループ カウンタを指定しない **FOR...NEXT** ループのネストの例です。

```
FOR I = 1 TO 2
    FOR J = 4 TO 5
        FOR K = 7 TO 8
            PRINT I, J, K
        NEXT
    NEXT
NEXT
```

出力例

```
 1              4              7
 1              4              8
 1              5              7
 1              5              8
 2              4              7
 2              4              8
 2              5              7
 2              5              8
```

EXIT FOR による FOR...NEXT ループの終了

EXIT FOR ステートメントを使うと、ループの終了条件が満たされる前に強制的にループを終了させることができます。**EXIT FOR** は、**FOR...NEXT** ループ中にいくつ指定してもかまいません。また、ループ中のどこに置いてもかまいません。

```
'乱数の平方根を計算し，それが 3 で割り切れるときにループを終了して，
'結果を表示します．
RANDOMIZE TIMER      '乱数ジェネレータを初期化します。
FOR I% = 1 TO 1000
    Root = SQR(RND)
    IF Root MOD 3 = 0 THEN EXIT FOR
NEXT                                 'カウンタは I%
MSGBOX STR$(I%) + " 回目に " + STR$(Root) + " を得ました。"
```

EXIT FOR は、最も内側の **FOR...NEXT** だけを終了させます。例えば、次のようにネストしたループを実行しているときに **EXIT FOR** を実行すると、最も内側のループ (ループ カウンタが J のループ) だけが終了します。

```
RANDOMIZE TIMER        '乱数ジェネレータを初期化します。
FOR I% = 1 TO 1000
    FOR J% = 1 TO 1000
        Root = SQR(RND)
        IF Root Mod 3 = 0 THEN EXIT FOR
    NEXT                            'カウンタは J%
    MSGBOX STR$(I%) + " 回目に " + STR$(Root) + " を得ました。"
NEXT                                'カウンタは I%
```

DO...LOOP ループ

DO...LOOP ループは、ループを実行する回数は決まっていないが、ループを終了する条件が明確なときに使います。ループを実行する回数があらかじめ決まっているときは、**FOR...NEXT** ステートメントを使ってください。**DO...LOOP** では、ループの終了条件を満たすまでステートメント ブロックを繰り返し実行します。

DO...LOOP は、真または偽のどちらで条件をテストすることもできるため、非常に柔軟性があります。また、真偽のテストはループの先頭でも末尾でも行うことができます。ループ条件を指定しないとループは終了しませんが、別の方法でループから抜け出ることは可能です。

DO...LOOP ステートメントには、条件が真の間ループする **DO WHILE...LOOP** と、条件が偽の間（真になるまで）ループする **DO UNTIL...LOOP** があります。

ループの先頭で条件をテストする方法

DO...LOOP ループの先頭で条件をテストするようにすると、ループが 1 度も実行されないこともあります。次の表は、ループの先頭でテストするときの構文を示します。

表 15.5 DO...LOOP ステートメントの構文と使用例：ループの先頭でテストする

構文	使用例

DO [{**WHILE** | **UNTIL**} *condition*]
[*statementblock-1*]
[**EXIT DO**
[*statementblock-2*]]
LOOP
	`DO WHILE A$ <> "Y"`
	`'A$ が "Y" でない間、ループを繰り`
	`'返し実行します。`
	`LOOP`
	`DO UNTIL A$ = "Y"`
	`'UNTIL...= を使って、上と同じ`
	`'ループを実現します。`
	`LOOP`

次の例では、ループの先頭でテストを行います。

```
'DO WHILE NOT...LOOP による方法
'ファイル 1 の最後に達するまで，ファイルから 1 行読み込んで画面に出力します．
DO WHILE NOT EOF(1)
    LINE INPUT #1, LineBuffer$
    PRINT LineBuffer$
LOOP

'DO UNTIL...LOOP による方法
'ファイル 1 の最後に達するまで，ファイルから 1 行読み込んで画面に出力します．
DO UNTIL EOF(1)
    LINE INPUT #1, LineBuffer$
    PRINT LineBuffer$
LOOP
```

ループの末尾で条件をテストする方法

DO...LOOP ループの末尾で条件をテストするようにすると、ループは少なくとも 1 回は実行されます。次の表は、ループの末尾でテストするときの構文を示します。

表 15.6　DO...LOOP ステートメントの構文と使用例：ループの終わりでテストする

構文	使用例

DO
[*statementblock-1*]
[**EXIT DO**
[*statementblock-2*]]
LOOP [{**WHILE** | **UNTIL**} *condition*]
	`DO`
	`'A$ が "Y" でない間、ループを繰り`
	`'返し実行します。`
	`LOOP WHILE A$ <> "Y"`
	`DO`
	`'UNTIL...= を使って、上と同じ`
	`'ループを実現します。`
	`LOOP UNTIL A$ = "Y"`

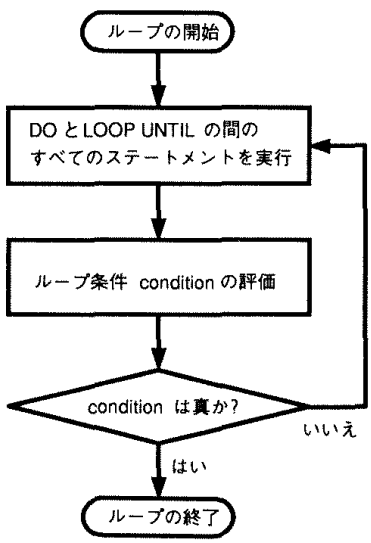


図 15.1　DO...LOOP UNTIL の論理図

EXIT DO による DO...LOOP ループの終了

これまでに示した **DO...LOOP** の例では、条件のテストをループの先頭または終わりで行っていましたが、**EXIT DO** ステートメントを使うと、ループの終了条件が満たされる前に強制的にループを終了させることができます。**EXIT DO** は、**DO...LOOP** ループ中にいくつ指定してもかまいません。また、ループ中のどこに置いてもかまいません。

次の例では、ファイルをオープンし、ファイルの終わりに達するか、ユーザーが指定したパターンが見つかるまで、ファイルを 1 行ずつ読み込みます。ファイルの終わりに達する前にパターンが見つかると、**EXIT DO** ステートメントを実行してループを終了します。

```
File$ = INPUTBOX$ ("検索対象のファイル:")
IF File$ = "" OR DIR$(File$) = "" THEN END

Pattern$ = INPUTBOX$ ("検索するパターン:")
IF Pattern$ = "" THEN END
OPEN File$ FOR INPUT AS #1

DO UNTIL EOF(1)       'ファイルの終わりに達すると, EOF(1) が
                      '真の値を返します.
    LINE INPUT #1, TempLine$
    IF INSTR(TempLine$, Pattern$) > 0 THEN
                      'パターンを含む最初の行を出力して、
                      'ループから抜けます。
        MSGBOX TempLine$
    EXIT DO
    END IF
LOOP
```

WHILE...WEND ループ

WHILE...WEND ステートメントは、**DO...LOOP** ステートメントと同じようにループを終了する条件が明確なときに使います。**WHILE...WEND** は、条件が真の間ループを繰り返します。**WHILE...WEND** ループには、**DO...LOOP** ループの **EXIT DO** のようにループを強制的に終了させる方法は用意されていません。また、条件は必ずループの先頭でテストします。**DO...LOOP** ループの方がより柔軟性に富んだプログラミングが可能なため、通常は **DO...LOOP** ループを使うことをおすすめします。ただしプログラミング環境でアプリケーションを実行するときは、**DO...LOOP** より **WHILE...WEND** の方がより高速です。実行可能 (.EXE) ファイルではこのような差はありません。次の表は **WHILE...WEND** ステートメントの構文と使用例を示します。

表 15.7 WHILE...WEND ステートメントの構文と使用例

構文	使用例
WHILE *condition* [*statementblock*] **WEND**	R$ = INPUTBOX$("最初の文字は") WHILE R$ <> "X" R$ = INPUTBOX$("次の文字は") WEND

```
'WHILE...WEND ではループの先頭で条件をテストするので，ループを少なくとも
'1 回実行するには，Response$ に "Y" を代入する必要があります．
Response$ = "Y"
WHILE UCASE$(Response$) = "Y"
  .
  .
  .
Response$ = INPUTBOX$ "もう 1 度"
WEND
'上と同じループを DO...LOOP を使って書き直すと次のようになります．
'ここでは，ループの最後で条件をテストします．
DO
  .
  .
  .
Response$ = INPUTBOX$ "もう 1 度"
LOOP WHILE UCASE$(Response$) = "Y"
```

16 プロシージャによる構造化プログラミング

プログラムを書く前にアプリケーション全体についての設計を行い、論理的に独立した部分を組み合わせてプログラムを作成することを構造化プログラミングと呼びます。Visual Basic は、構造化プログラミングをサポートしています。小さなタスクを行ういくつかのステートメントが集まってプロシージャを構成し、プロシージャが集まって、より大きなタスクを実現するモジュール (ソース ファイル) を構成します。さらに、大規模なアプリケーションでは、複数のモジュールから 1 つのアプリケーションを構成します。適切な設計を行って汎用的なプロシージャを作成しておけば、それらのプロシージャを集めたモジュールは、別のアプリケーションでそのまま利用することができます。このようにすると、Visual Basic 自身を強化，拡張するような効果があります。

Visual Basic は、**FUNCTION** と **SUB** の 2 種類のプロシージャをサポートしています。**FUNCTION** プロシージャは結果として値を返し、Visual Basic が最初から用意している関数のように、式の中に記述することによって呼び出します。**SUB** プロシージャは、ステートメントと同じように扱うことができ、値を返しません。イベントの発生によって自動的に呼び出されるイベント プロシージャも、**SUB** プロシージャの一種です。

ここでは、プロシージャとモジュールによって、大きなプログラムを小さな論理単位に分割し、アプリケーションを構造化する方法について説明します。

16.1 プロシージャ

“プロシージャ” という用語は、**SUB...END SUB** および **FUNCTION...END FUNCTION** 構造だけを指します。**GOSUB...RETURN** と **DEF FN...END DEF** 構造も従来の Basic との互換性のために残されていますが、これらはプロシージャとは呼びません。

イベント プロシージャ

“イベント プロシージャ” は、イベントの発生によって自動的に呼び出される **SUB** プロシージャで、フォーム モジュールに置かれます。オブジェクトと、それに対応するイベントごとに、イベント プロシージャを定義します。イベント プロシージャからは、同じフォーム モジュールに含まれる他のプロシージャを呼び出すことができます。また、同じプロジェクトに含まれるコード モジュール中のプロシージャも同様に呼び出すことができます。ただし、イベント プロシージャはフォームごとに定義されるため、別のフォーム モジュールのプロシージャを直接呼び出すことはできません。

イベント プロシージャの作成と使い方については、第 3 章「イベントドリブン アプリケーションの作成」を参照してください。

ジェネラル プロシージャ

イベント プロシージャ以外のすべてのプロシージャを、“ジェネラル プロシージャ”と呼びます。Quick Basic や Microsoft Basic のプロシージャは、すべてジェネラル プロシージャです。通常、繰り返し実行するタスクを、ジェネラル プロシージャにまとめます。例えば、複数のテキスト ボックスで、フォーカスが別のコントロールに移ったときに内容を確認したい場合、それぞれのテキスト ボックスの LostFocus イベント プロシージャに確認のためのコードを記述することもできます。しかし、そのコードを 1 つのプロシージャにしておき、それぞれの LostFocus イベント プロシージャから呼び出すようにすると、同じ処理を重複して記述しなくてすみます。このように、プロシージャを使うと、それぞれの目的にあった効率的なプログラミングを行うことができます。

プロシージャを使ってプログラミングすると、次のような利点があります。

- プログラムを独立した論理単位に分割できます。プログラム全体をデバッグするときよりも、プロシージャ単位でデバッグする方が簡単です。
- 1 度記述したコードを再利用できます。あるプログラムで使われているプロシージャは、ほとんど変更せずに、他のプログラムの構成単位として使うことができます。
- プロシージャで使うローカル変数は、プロシージャを実行しているときのみ存在するので、大きなプログラムを書くときにメモリを節約できます。

また、汎用的なプロシージャを集めて、Quick ライブラリに組み込むこともできます。Quick ライブラリは、Visual Basic の起動時にロードされる特殊なファイルです。Quick ライブラリをロードすると、アプリケーションからライブラリ中のプロシージャを、Visual Basic が最初から用意しているステートメントや関数と同じように呼び出すことができます。Quick ライブラリについては、第 18 章「Quick ライブラリの作成」を参照してください。

▼ ジェネラル プロシージャを作成するには

1. プログラミング環境で、[編集] メニューの [新規 SUB プロシージャ] または [新規 FUNCTION プロシージャ] コマンドを選択します。表示されるダイアログ ボックスに対して、プロシージャ名を入力してください。または、コード ウィンドウに例えば次のようなプロシージャの最初の行を入力して [ENTER] キーを押すと、自動的にプロシージャが作成されます。

   ```
   SUB New
   ```

Visual Basic は、アクティブになっているコード ウィンドウの内容をクリアし、アクティブ ウィンドウにプロシージャのテンプレート (最初の行と最後の行) を表示します。

```
SUB New()

END SUB
```

2. この 2 つのステートメントの間に、プロシージャの内容を入力します。

複数のモジュールからなるプログラムでのプロシージャの使い方

複数のモジュールを使うときは、コードとデータのスコープ (通用範囲) について理解しておく必要があります。例えば、あるコード モジュールで **FUNCTION** プロシージャを定義し、それを別のモジュールから呼び出すときは、呼び出し側のモジュールで **DECLARE** ステートメントを使って、呼び出すプロシージャを宣言しなければなりません。**DECLARE** ステートメントがないと、関数の呼び出しは配列に対する参照と解釈されてしまいます。ただし Visual Basic は、モジュールの保管時に **DECLARE** ステートメントを自動的に生成する機能を持っているため、通常はこのような問題は起こりません。

フォーム モジュールから、そのモジュールに含まれるフォームのプロパティを参照するときは、プロパティ名を指定するだけでかまいません。フォームの指定は省略できます。例えば、Form_Load イベント プロシージャでは、次のようにオブジェクト名を省略してフォームの幅を設定できます。

```
Width = 25
```

別のモジュールに含まれるフォームの幅を設定するには、プロパティ名の前に必ずオブジェクト名を指定しなければなりません。

```
Form2.Width = 25
```

また、同じモジュールに含まれるフォーム上のコントロールのプロパティを参照するには、プロパティ名の前に必ずコントロール名を指定します。

```
Label1.Width = 30
```

別のフォーム上のコントロールを参照するには、コントロール名の前に、さらにそのコントロールが配置されているフォーム名を指定します。

```
Form2.Label1.Width = 30
```

プロシージャへのデータの引き渡し

プロシージャを呼び出すとき、プロシージャに情報を引き渡すことができます。プロシージャは、アプリケーション中で何回でも呼び出せますが、呼び出すたびに異なるデータを引き渡せます。また、1 度の呼び出しで複数の値を受け渡すことができます。データを引き渡すには、そのプロシージャが値を受け取るパラメータを指定してプロシージャを定義します。指定されたパラメータを、パラメータのリストとも呼びます。また、プロシージャを呼び出すときに、プロシージャに渡す値を引数として指定します。これを、引数のリストと呼びます。次の例では、**SUB** プロシージャ Compare を 2 回呼び出しますが、それぞれ異なる引数を指定しています。

```
CALL Compare(4, 5)          'この呼び出しでは，引数は 4 と 5 です．
CALL Compare(7, 2)          'この呼び出しでは，引数は 7 と 2 です．
END
SUB Compare(A, B)                    'Compare を定義するときは，
    IF A < B THEN SWAP A, B          'パラメータ A と B を指定します．
END SUB
```

プロシージャ内では、引き渡された変数の値を変更できます。詳しくは、この章の「参照による引き渡しと値による引き渡し」を参照してください。

ローカル変数とグローバル変数

プロシージャ内で定義されるすべての変数は、特に指定しなければプロシージャに対してローカルです。プログラム中の他の部分で使われている変数と同じ名前を持っていても、異なる変数として扱われます。いろいろなプロシージャで同じ変数名を使っても問題はありません。ただし、必要に応じて変数がより広いスコープ (通用範囲) を持つように指定することができます。プロシージャとグローバル変数については、「変数の共有」を参照してください。

プロシージャの定義

Visual Basic では、一般に次のような構文を使ってプロシージャを定義します。

[STATIC]{SUB | FUNCTION} *procedurename* [(*parameterlist*)] **[AS** *type*]

statementblock-1

[EXIT {SUB | FUNCTION}]. . .

[*statementblock-2*]. . .

END {SUB | FUNCTION}

プロシージャ定義のそれぞれの部分について説明します。

定義部分	説明
STATIC	ローカル変数を **STATIC** として扱います。このときローカル変数の値は、次にプロシージャが呼び出されるまで保持されます。**STATIC** キーワードを省略すると、ローカル変数はデフォルトで自動変数になります。このときローカル変数の値は、プロシージャが呼び出されるたびに 0 または null に初期化されます。
procedurename	プロシージャ名 (最大 40 文字) を指定します。この名前は、変数と同じ名前付け規約に従っていなければなりません。複数のプロシージャに同じ名前を付けることはできません。イベント プロシージャの名前は、オブジェクト名 (*FormName* または *CtlName*) の後にアンダースコア文字とイベント名を付けたもの (*CltName*_Click など) になります。
parameterlist	パラメータとなる変数を指定します。複数の変数を指定するときはカンマで区切ります。この指定は、プロシージャに引き渡される引数の個数とデータ型を示します。呼び出し側で指定された引数は、対応するパラメータに引き渡されます。プロシージャ内では、引数は *parameterlist* で指定した変数名によって参照されます。
AS *type*	**FUNCTION** プロシージャで、戻り値のデータ型を指定します。プロシージャ名の後に型宣言文字を付けるときは指定できません。型宣言文字を付けないときに **AS** *type* の指定を省略すると、**FUNCTION** の戻り値の型は、そのモジュールのデフォルトのデータ型になります。
EXIT SUB **EXIT FUNCTION**	プロシージャの実行を終了します。**EXIT** は、ある条件を満たしたときにプロシージャを終了したり、ローカル エラー処理の前にプロシージャから抜けるときにも使うことができます。

例えば、**FUNCTION** プロシージャが文字列を返すときは、その名前に型宣言文字のドルマーク ($) を付けるか、**AS STRING** と指定して定義するか、**DEFSTR** ステートメントで文字列型として宣言しておかなければなりません。

プロシージャ内では、次に示すステートメントは使用できません。

- **DEF FN...END DEF**, **FUNCTION...END FUNCTION**, **SUB...END SUB**
- **COMMON**
- **DECLARE**
- **DIM SHARED**
- **OPTION BASE**
- **TYPE...END TYPE**
- **STACK** *n*

注　意　プロシージャ定義をネストしたり、プロシージャ内で **DEF FN** 関数を定義することはできません。ただし、プロシージャから別のプロシージャや **DEF FN** 関数を呼び出すことは可能です。

FUNCTION プロシージャの呼び出し

FUNCTION プロシージャは、組み込み関数と同様に、式の中で名前を使って参照します。例えば次の各行は、IntegerPower という名前の **FUNCTION** プロシージャを呼び出します。

```
MSGBOX "答は: " + STR$(IntegerPower(12, 6))
X& = IntegerPower(n&, 7)
IF IntegerPower(a&, b&) = 10 THEN MSGBOX "範囲外です"
```

FUNCTION プロシージャが返す値は、プロシージャの中で **FUNCTION** プロシージャ名 (ユーザー定義関数名) に値を代入することによって指定します。

```
STATIC FUNCTION IntegerPower(X AS LONG, Y AS LONG) AS LONG
    PowerVal& = 1
    FOR I& = 1 TO Y&
        PowerVal& = PowerVal& * X&
    NEXT I&
    IntegerPower = PowerVal&      ' 戻り値は関数名に代入されます.
END FUNCTION
```

SUB プロシージャの呼び出し

SUB プロシージャは、式の中で名前を使って呼び出すことはできません。また、実行の結果として値を返しません。

SUB プロシージャを呼び出すには、次の 2 つの方法があります。

- **CALL** ステートメントでプロシージャ名を指定する

  ```
  CALL PrintMessage (100, "Test")
  ```

- プロシージャ名を、独立したステートメントのように記述する

  ```
  PrintMessage 100, "Test"
  ```

2 番目の、**CALL** キーワードを指定しない方法では、**SUB** プロシージャに引き渡す引数を、カッコで囲まずに指定しなければなりません。また、プロシージャを呼び出す前に、そのプロシージャを **DECLARE** ステートメントで宣言しておく必要があります。

```
DECLARE SUB PrintMessage (X%, Y$)
  .
  .
  .
PrintMessage 100, "Test"
```

Visual Basic は、モジュールやアプリケーション全体をセーブするとき、必要な場所に **DECLARE** ステートメントを自動的に挿入します。ただし、フォーム モジュールに含まれるプロシージャは他のモジュールから呼び出すことができないので、これらに対しては **DECLARE** ステートメントは生成されません。また、イベント プロシージャは Visual Basic が自動的に呼び出すため、**DECLARE** ステートメントは必要ありません。

16.2　プロシージャへの引数の引き渡し

ここでは、パラメータと引数の違い，引数をプロシージャに引き渡す方法，引数のデータ型と個数をチェックする方法について説明します。

パラメータと引数

プロシージャに引数を引き渡し方法を説明する前に、“パラメータ”と“引数”の違いについて説明しておきます。

パラメータ	引数
SUB または **FUNCTION** ステートメント，またはプロシージャの **DECLARE** ステートメントで指定する変数名	プロシージャを呼び出すときに、**SUB** または **FUNCTION** に引き渡される定数，変数，式

パラメータは、仮引数や形式パラメータとも呼ばれ、引数は実引数とも呼ばれます。プロシージャを定義するとき、パラメータは引数を受け取るための仮の名前として使います。図 16.1 に示すように、プロシージャが呼び出されると、パラメータ リストに指定された変数に引数が順に割り当てられます。最初のパラメータは最初の引数、2 番目のパラメータは 2 番目の引数というように、それぞれのパラメータが対応する引数を受け取ります。

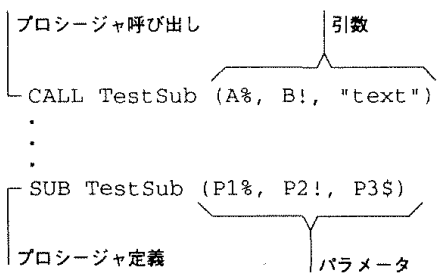


図 16.1　パラメータと引数

図 16.1 は、引数とパラメータの関係を示します。引数リストとパラメータ リストに、同じ変数名を指定する必要はありません。ただし、パラメータの個数と引数の個数は同じでなければなりません。また、対応する引数とパラメータのデータ型 (整数，長整数，単精度浮動小数点数，倍精度浮動小数点数，通貨型，文字列型) は同じでなければなりません (引数とパラメータのデータ型と個数をチェックする方法については、「DECLARE による引数のチェック」を参照してください)。引数のデータ型が、対応するパラメータのデータ型と一致するように強制的に変換される場合もあります。例えば、引数に整数の定数を指定したとき、対応するパラメータが長整数や浮動小数点

数なら、自動的にそれらのデータ型に変換されます。ただし、実際に渡される値の精度は、引数に指定された低い精度のままです。

パラメータ リストは次のいずれかの項目からなり、それぞれカンマで区切ります。

- 固定長文字列以外の有効な変数名

  例えば、X$ と X AS STRING は、どちらも可変長文字列であり、パラメータとして指定できます。ただし、X AS STRING * 10 は、10 バイトの固定長文字列を示すため、パラメータとして指定できません (引数には、固定長文字列も指定できます)。

- 配列変数名の後に、空のカッコを付けたもの
- 固定長文字列の配列

引数リストは次のいずれかの項目からなり、それぞれカンマで区切ります。

- 定数
- 式
- 有効な変数名 (ユーザー定義型を含みます)
- 配列変数名の後に、空のカッコを付けたもの

次の **SUB** ステートメントでは、さまざまな種類のパラメータを指定しています。

```
SUB TestSub (A%, Array(), RecVar AS RecType, Cs$)
```

最初のパラメータ A% は整数です。2 番目のパラメータ Array() は配列で、データ型はそのモジュールのデフォルトのデータ型になります。3 番目のパラメータ RecVar はユーザー定義型である RecType 型の変数です。4 番目のパラメータ Cs$ は文字列です。

次の例は、CALL TestSub の行で TestSub プロシージャを呼び出して、対応するデータ型を持つ 4 つの引数を引き渡します。

```
TYPE RecType
    Rank AS STRING * 12
    SerialNum AS LONG
END TYPE

DIM RecVar AS RecType
DIM AR(1 TO 25) AS INTEGER

CALL TestSub(X%, AR(), RecVar, "Daphne")
```

定数と式の引き渡し

定数 (文字列または数値) は、引数としてプロシージャに引き渡すことができます。文字列パラメータには文字列定数を、数値パラメータには数値定数を引き渡さなければなりません。

引数リストで指定する数値定数の型が、**SUB** または **FUNCTION** ステートメント中の対応するパラメータと同じデータ型でなければ、強制的に適切な型に変換されます。次に例を示します。

```
CALL Test(4.6, 4.1)

SUB Test (x%, y%)
    Msg$ = "値は " + STR$(x%) + " と " + STR$(y%)
    MSGBOX Msg$
END SUB
```

上のプログラムを実行すると、**MSGBOX** ダイアログ ボックスに次のように表示されます。

```
値は 5 と 4
```

変数や定数を組み合わせた式も、プロシージャに引き渡すことができます。定数の場合と同じように、数式の型がパラメータの型と一致しないときは、強制的にプロシージャで定義されたパラメータの型に変換されます。

単純変数の引き渡し

引数リストまたはパラメータ リストでは、次の 3 つの方法で単純変数のデータ型を宣言できます。

- 変数名に型宣言文字 (%, &, !, #, @, $) のいずれかを付けます。
- パラメータ名の後で **AS** キーワードを使って指定します。次に例を示します。

  ```
  SUB MySub(Param AS LONG)
  ```

- **DEF***type* ステートメントでデフォルトのデータ型を設定します。

いずれの場合でも、引数リストとパラメータ リストで対応する変数は、同じ型でなければなりません。

配列変数の引き渡し

配列変数のすべての要素をプロシージャに引き渡すときは、引数リストとパラメータリストで、名前の後ろに空のカッコを付けた配列変数名を指定します。

```
DIM Values(1 TO 12) AS INTEGER

'プロシージャに配列を引き渡すときは，配列変数名の後に空のカッコを指定する
'ことに注意してください．
CALL ChangeArray(1, 12, Values())
```

```
'パラメータ P の後に空のカッコを付けていることに注意してください.
SUB ChangeArray(Min%, Max%, P() AS INTEGER)
    FOR I% = Min% TO Max%
        P(I%) = I% ^ 3
    NEXT I%
END SUB
```

プロシージャに対して、配列変数の個々の要素を引き渡すこともできます。配列変数の特定の要素を引き渡すには、配列変数名の後ろのカッコの中に、要素を示す添字を指定してください。次の例では、配列の要素 (4, 2) を **SUB** プロシージャ SqrRootVal に引き渡します (このプロシージャでは、パラメータとして指定した変数の値を変更しているため、呼び出し側の配列の要素の値も変更されることに注意してください)。

```
DIM Array(1 TO 5,1 TO 3)

Array(4,2) = VAL(INPUTBOX$("値を入力してください"))
CALL SqrRootVal(Array(4,2))

SUB SqrRootVal(A)
    A = SQR(ABS(A))
END SUB
```

LBOUND 関数と **UBOUND** 関数は、プロシージャに引き渡された配列のサイズを調べるための関数です。**LBOUND** 関数は配列の添字の最小値を返します。また、**UBOUND** 関数は最大値を返します。これらの関数を使うと、前の ChangeArray の例のように、配列の各次元の最大値と最小値をプロシージャに引き渡す必要はありません。

次の例では、**LBOUND** 関数を使って、配列 A の各次元の添字の最小値で変数 Row と Col を初期化します。また、**UBOUND** 関数を使って、配列中の要素数だけ **FOR** ループを実行するようにします。

```
STATIC SUB PrintOut(A() AS INTEGER)
    FOR Row = LBOUND(A,1) TO UBOUND(A,1)
        FOR Col = LBOUND(A,2) TO UBOUND(A,2)
            PRINT A(Row,Col)
        NEXT Col
    NEXT Row
END SUB
```

1 次元の配列を引き渡すときに、**LBOUND** 関数を使って 2 番目の次元の最小値を調べようとするとエラーになります。

ユーザー定義型変数の引き渡し

ユーザー定義型の変数 (レコードとも呼びます) を引き渡すには、次のようにします。

▼ ユーザー定義型変数全体を引き渡すには

1. ユーザー定義型を定義します。

```
TYPE StockItem
    PartNumber AS STRING * 6
    Description AS STRING * 20
    UnitPrice AS SINGLE
```

```
    Quantity AS INTEGER
END TYPE
```

2. この型を持つ変数を宣言します。

```
DIM StockRecord AS StockItem
```

3. プロシージャ定義では、変数と同じ型を持つパラメータを指定します。

```
STATIC SUB FindRecord (RecordVar AS StockItem)
  .
  .
  .
END SUB
```

4. プロシージャを呼び出して、宣言した変数をプロシージャに引き渡します。

```
CALL FindRecord(StockRecord)
```

宣言を行う位置については、「TYPE ステートメントと DECLARE ステートメントの位置関係」を参照してください。

ユーザー定義型変数を引き渡すときには注意が必要です。パラメータとしてユーザー定義型を指定すると、自動生成された **DECLARE** ステートメントでは、データ型は **ANY** となります。**ANY** は、データ型が不定であることを示し、プロシージャを呼び出すときの型チェックが行われません。したがって、**DECLARE** ステートメント中の **ANY** キーワードを、**TYPE...END TYPE** ステートメントで実際に定義した適切なデータ型で置き換えるようにしてください。この置き換えを行わないと、引数の型のチェックを行うときにエラーを検出できません。

ユーザー定義型変数のそれぞれの要素をプロシージャに引き渡すには、引数リストで要素名 (*recordname.elementname*) を指定します。次に例を示します。

```
DECLARE SUB PrintTag (Desc AS STRING, Price AS CURRENCY)
TYPE StockType
    PartNumber AS STRING * 6
    Descrip AS STRING * 20
    UnitPrice AS CURRENCY
    Quantity AS INTEGER
END TYPE

DIM StockItem AS StockType
CALL PrintTag (StockItem.Descrip, StockItem.UnitPrice)
  .
  .
  .
END

SUB PrintTag (Desc AS STRING, Price AS CURRENCY)
  .
  .
  .
END SUB
```

関数呼び出しを含む式の引き渡し

関数呼び出しも式の一種であり、関数呼び出しや関数呼び出しを含む式をプロシージャに引き渡すことができます。次の例では、MyFunction が関数です。

```
CALL MySub(MyFunction(100) + 3)
```

未知の型の引数の引き渡し

プロシージャのパラメータの定義で、データ型として **AS ANY** と指定しているときは、すべての型の引数を引き渡すことができます。この場合、プロシージャを呼び出すときの引数のデータ型のチェックが行われません。**AS ANY** を使ってパラメータを宣言するときも、引き渡す引数の個数がパラメータの個数と一致しなければならないことに注意してください。

フォームとコントロールの引き渡し

SUB または **FUNCTION** ステートメントで次のようにパラメータを定義すると、プロシージャは、パラメータとしてフォームまたはコントロールを受け取ることができます。

```
SUB ClearBoxes(TheListBox as CONTROL, TheForm AS FORM)
    TheForm.BackColor = 9
    FOR ind% = 0 TO TheListBox.ListCount -1
        TheListBox.REMOVEITEM 0          'それぞれの項目を削除します.
    NEXT
END SUB
```

このようにすると、フォームやコントロール全体をプロシージャに引き渡すことができます。この場合、プロシージャ内でプロパティの設定を変更すると、呼び出し側にも影響します (参照による引き渡し)。また、Caption プロパティなどのフォームやコントロールの各プロパティを、プロシージャに渡すこともできます。この場合、Caption プロパティなどの文字列を示すものはパラメータとして文字列変数を指定し、Visible プロパティなどの数値を示すものはパラメータとして数値変数を指定します。また、引数としてプロパティを指定するとき、カッコで囲んで式として指定しなければなりません。この場合、プロシージャ内でパラメータとして指定されている変数の値を変更しても、実際のプロパティの値には影響しません。

16.3　参照による引き渡しと値による引き渡し

プロシージャに引数を引き渡すとき、参照による引き渡しと値による引き渡しの 2 つの方法があります。

引数の参照による引き渡し

引数を参照によって引き渡すと、呼び出し側とプロシージャで、同じ変数に対して操作が行われます。プロシージャ内でパラメータに指定した変数の値を変更すると、呼び出し側で引数に指定した変数の値も変更されることになります。

Visual Basic の引数はデフォルトで、参照によって引き渡されます。変数を参照によって引き渡すとき、内部的には、プロシージャのパラメータが受け取る情報は、引数として指定された変数のアドレスになります。

- それぞれの変数の値は、あるアドレス (メモリ中の特定の位置) に格納されています。
- 参照によってプロシージャに変数が引き渡されると、そのプロシージャは、それぞれの変数のアドレスを受け取ります。このとき、引き渡された変数のアドレスと、プロシージャ内の対応するパラメータのアドレスが同じになります。
- このため、プロシージャでパラメータとして指定した変数の値を変更すると、呼び出し側で引数として指定した変数の値も変化します。

```
AnyVal% = 7
CALL Doubler(AnyVal%)
MSGBOX STR$(AnyVal%)

SUB Doubler(TheNumber AS INTEGER)
    TheNumber = TheNumber * 2
END SUB
```

このプログラムを実行すると、呼び出し側の AnyVal% と、プロシージャ内の TheNumber は同じアドレスを参照し、1 つの値を共有します。呼び出し側では、受け取った TheNumber の値である 7 を 2 倍して、再び変数に代入しているため、**MSGBOX** ダイアログ ボックスには 14 と表示されます。

引数の値による引き渡し

引数の値による引き渡しでは、引数のアドレスでなく、引数の値そのものが引き渡されます。プロシージャ内でパラメータに指定した変数の値を変更しても、呼び出し側で引数に指定した変数には影響しません。

値による引き渡しの場合、プロシージャが呼び出されると、引数に指定された値はメモリ上の別の場所にコピーされ、この一時的な場所のアドレスがパラメータに引き渡されます。プロシージャは、呼び出し側の変数のアドレスを持たないため、呼び出し側の変数の値を変更することはできません。すべての操作は、引数に指定された変数のコピーに対して実行されます。

変数を値によって引き渡す方法の 1 つに、変数をカッコで囲んで式として扱う方法があります。次の例では、プロシージャでどのような変更を行っても、呼び出し側の変数の値は変化しません。

```
CALL AnySub("名前は?:", 57, a%+57, (b%), b%-a%, (AnyString$))

SUB AnySub(Prompt$, Age%, Sum%, HRate%, Diff%, AnyString$)
  .
  .
  .
END SUB
```

変数を値によって引き渡すもう1つの方法は、**BYVAL** キーワードを指定することです。**BYVAL** を指定した変数は、その変数の値が格納されているアドレスではなく、値そのものがプロシージャに引き渡されます。次に例を示します。

```
SUB AgeDiscount(BYVAL Age%, BYVAL Price@, BYVAL CustNum&)
    IF MSGBOX("値引き価格を見ますか?", 4) = 6 THEN
        Age% = Age% \ 2
        Price@ = Price@ - (Price@ * (Age% *.01))
        Msg$ = "値段は: $" + STR$(Price@) + CHR$(13) + CHR$(10)
        Msg$ = Msg$ + "その値段で買いますか?"
        IF MSGBOX(Msg$, 4) = 6 THEN
            CALL FillOrder (Price@, CustNum&)
        END IF
    END IF
END SUB
```

次に示す AgeDiscount プロシージャの呼び出しは、通常の呼び出しと同じように見えますが、引数に指定された変数の値は、プロシージャでの操作の影響を受けません。AgeDiscount を呼び出すと、引数の実際の値がプロシージャのパラメータに割り当てられます。

```
CALL AgeDiscount (CustAge%, FullRetail@, CustNumber&)
```

Visual Basic では、引数としてプロパティが指定される可能性がある場合に、パラメータに **BYVAL** を指定することができます。この場合、プロシージャ中でパラメータの値を操作しても、フォームやコントロール自身には影響を与えません。呼び出されるプロシージャでフォームやコントロールのプロパティを直接操作できるようにするには、フォーム全体またはコントロール全体を引き渡してから、必要なプロパティに対して操作を行います。次のようにプロシージャを定義すると、プロシージャ内で変数 TheTop の値を変更しても、フォームの位置は変化しません。

```
SUB FindSpace(BYVAL TheTop AS INTEGER)
    IF TheTop =< 5 THEN TheTop = TheTop + 5
END SUB

CALL FindSpace(Form1.Top)
```

この例では、フォームの Top プロパティをプロシージャに引き渡しています。Top プロパティは数値で表すことができるため、プロシージャ側は整数として受け取っています。**BYVAL** キーワードによって、パラメータはプロパティそのものではなく、プロパティのそのときの値を受け取るため、プロシージャでパラメータに指定した変数の値を変更しても、プロパティ自身の設定は変わりません。なお、個々のプロパティを参照によってプロシージャに引き渡すことはできません。

注　意　**BYVAL** キーワードは、パラメータに対して指定したときに意味を持ちます。引数に対して **BYVAL** キーワードを指定することもできますが、パラメータに対して **BYVAL** を指定してあれば、引数に対して **BYVAL** を指定する必要はありません。パラメータに対して **BYVAL** を指定していないときに引数に対して **BYVAL** を指定すると、プロシージャを呼び出すときの引数のデータ型のチェックでエラーが発生します。そのプロシージャが **DECLARE** ステートメントで宣言されていないときは、その引数を値によって引き渡そうとしますが、結果は予測できません。

配列，ユーザー定義型の引き渡しの方法

配列変数やユーザー定義型の変数は、それ自体を値によって引き渡すことはできず、必ず参照によって引き渡します。配列変数やユーザー定義型変数の名前をカッコで囲んでも、式の中で評価できないため、エラーになります。配列変数やユーザー定義型変数をプロシージャに渡したいときは、全体を参照で引き渡すようにするか、呼び出し前に、プロシージャで必要な要素だけを別の変数にコピーして、その値を渡すなどの方法を取ります。

引数に式として指定できない項目

ユーザー定義型の変数，配列変数，フォーム，コントロールは、それ自身を式の中に含めることができません。これらをプロシージャに引き渡すには、参照で渡します。ただし、ユーザー定義型変数や配列変数の各要素，フォームやコントロールのプロパティは、式として指定して値で引き渡すことができます。

16.4　DECLARE による引数のチェック

DECLARE ステートメントであらかじめプロシージャを宣言しておけば、そのプロシージャが定義されているモジュール以外のモジュールから、プロシージャを呼び出すことができます。また、Quick ライブラリに組み込まれているプロシージャを呼び出す場合も、**DECLARE** ステートメントが必要です。

プログラミング環境では、プロジェクトを保管するときに、必要な **DECLARE** ステートメントが自動的に判断されて挿入されます。**DECLARE** ステートメントでは、プロシージャ名の他に、引数の個数とデータ型が宣言されます。この部分の指定は、**SUB** または **FUNCTION** ステートメントで指定するパラメータ リストと同じです。**DECLARE** ステートメントの構文については、『リファレンス』またはオンライン ヘルプを参照してください。

DECLARE ステートメントのパラメータ リストは、プロシージャに引き渡される引数の型チェックに利用されます。引数に変数を使ってプロシージャが呼び出されると、これらの引数の個数や型が **DECLARE** ステートメント中のパラメータの個数や型と一致するかどうかチェックされます。一致しないときはエラーになります。

ユーザー定義型のパラメータを持つプロシージャを定義して呼び出すときも、**DECLARE** ステートメントは自動的に生成されます。ただし、**SUB** または **FUNCTION** ステートメントで、パラメータのデータ型として適切なユーザー定義型を指定しているときでも、自動生成される **DECLARE** ステートメントでは、データ型は **AS ANY** として定義されます。このとき引数の型チェックを有効にするには、この **ANY** キーワードをユーザー定義型の名前 (**TYPE...END TYPE** ステートメントで定義された名前) で置き換えなければなりません。

次の例では、GetInput$ に引き渡す引数がないので、**DECLARE** ステートメントには空のカッコだけが指定されています。

```
DECLARE FUNCTION GetInput$ ()
TheName$ = GetInput$

FUNCTION GetInput$ ()
    GetInput$ = INPUTBOX$("名前を入力してください: ")
END FUNCTION
```

次の例では、GetInput$ に文字列引数を引き渡すので、**DECLARE** ステートメントにパラメータ リストが指定されています。

```
DECLARE FUNCTION GetInput$ (Prompt$)
TheName$ = GetInput$("名前を入力してください: ")

FUNCTION GetInput$ (Prompt$)
    GetInput$ = INPUTBOX$(Prompt$)
END FUNCTION
```

次の例は、ユーザー定義型のパラメータを持つプロシージャ定義と、このプロシージャの呼び出しで生成される **DECLARE** ステートメントを示します。

```
DECLARE SUB GetRecord (NewRecord AS ANY)

TYPE NewRecord
    UserName AS STRING * 30
    AccountBalance AS CURRENCY
END TYPE
DIM MyRecord AS NewRecord

CALL GetRecord(MyRecord)
SUB GetRecord (ARecord AS NewRecord)
  .
  .
  .
END SUB
```

DECLARE ステートメントの **ANY** キーワードは、適切なユーザー定義型の名前 (この場合は NewRecord) で置き換えるようにしてください。

DECLARE ステートメントが自動生成されない場合

状況によっては、**DECLARE** ステートメントが自動生成されないことがあります。

イベント プロシージャはフォーム モジュールの外側から呼び出すことができないので、イベント プロシージャに対する **DECLARE** ステートメントは生成されません。

SUB プロシージャに対する **DECLARE** ステートメントは、プロジェクトを保管するときに、プロシージャの定義と呼び出しの両方が存在しなければ自動生成されません。**SUB** プロシージャが定義されているモジュールがロードされていないときは、そのプロシージャに対する **DECLARE** ステートメントは自動生成されません。ただし、**CALL** ステートメントを使って **SUB** プロシージャを呼び出す場合は、**DECLARE** ステートメントは必要ありません。

FUNCTION プロシージャに対する **DECLARE** ステートメントが自動生成されるのは、**FUNCTION** プロシージャの定義と呼び出しが同じモジュール内で行われているときだけです。**FUNCTION** プロシージャに対する **DECLARE** ステートメントは、その **FUNCTION** プロシージャが別のモジュールで定義されている場合、ロードされているかどうかにかかわらず、自動生成されません。適切な **DECLARE** ステートメントが存在しない状態で呼び出そうとすると、**FUNCTION** プロシージャの呼び出しではなく、変数や配列変数の要素に対する参照と見なされてしまうため、正しく処理されません。**FUNCTION** プロシージャを呼び出すモジュールでは、適切な **DECLARE** ステートメントをプログラマが記述しなければなりません。

Quick ライブラリに組み込まれているプロシージャに対する **DECLARE** ステートメントも、自動的には生成されません。**DECLARE** ステートメントを含むインクルードファイルを作成し、**$INCLUDE** メタコマンドを使って各モジュールでインクルードすると便利です。

Visual Basic 以外の環境でのプログラム開発

プログラミング環境以外で、市販のテキスト エディタなどを使ってコードを入力し、BC および LINK コマンドでコンパイルするときは、次の場所に **DECLARE** ステートメントを挿入してください。

- **FUNCTION** プロシージャを呼び出すすべてのモジュールの先頭

```
DECLARE FUNCTION Hypot! (X!, Y!)
X! = VAL(Text1.Text): Y! = VAL(Text2.Text)
MSGBOX Str$(Hypot!(X!, Y!))
END

FUNCTION Hypot! (A!, B!)
    Hypot! = SQR(A! ^ 2 + B! ^ 2)
END FUNCTION
```

- **CALL** ステートメントを使わずに **SUB** プロシージャを呼び出すすべてのモジュールの先頭

```
DECLARE SUB ShowAndTrim (A!, B!)
X! = VAL(Text1.text) : Y! = VAL(Text2.text)
ShowAndTrim X!, Y!          '引数をカッコで囲まないことに
                            '注意してください.
END

SUB ShowAndTrim (A!, B!)
    '数値を文字列に変換し, 2 番目の数値の前の空白を削除して出力します.
    MSGBOX STR$(A!) + LTRIM$(STR$(B!))
END SUB
```

CALL ステートメントを使って **SUB** プロシージャを呼び出すときは、あらかじめ **SUB** プロシージャを宣言しておく必要はありません。

```
A$ = "466"
B$ = "123"
CALL ShowString(A$, B$)
END

SUB ShowString (X$, Y$)
    Msg$ = X$ + Y$
    MSGBOX Msg$
END SUB
```

- 別のモジュールで定義されている **SUB** または **FUNCTION** プロシージャを呼び出す、すべてのモジュールの先頭

次に示すように、プロシージャにパラメータがないときは、**DECLARE** ステートメントのプロシージャ名の後に必ず空のカッコを付けてください。

```
DECLARE FUNCTION GetHour$ ()
MSGBOX GetHour$
END

FUNCTION GetHour$ ()
    GetHour$ = LEFT$(TIME$,2)
END FUNCTION
```

DECLARE ステートメントが置けるのはモジュール レベルだけで、プロシージャの内部 (プロシージャ レベル) には記述できないことに注意してください。**DECLARE** ステートメントは、それが記述されているモジュール全体に対して有効です。

TYPE ステートメントと DECLARE ステートメントの位置関係

Visual Basic は、**DECLARE** ステートメントを自動生成するときに、モジュールの最初の行に挿入します。パラメータにユーザー定義型の変数が含まれるときは、このユーザー定義型を定義する **TYPE...END TYPE** ステートメントは、ユーザー定義型への参照を含む **DECLARE** ステートメントより前に置かなければなりません。したがって、場合によっては **TYPE...END TYPE** ステートメントを Visual Basic が生成する **DECLARE** ステートメントの前に移動しなければならないことがあります。

インクルード ファイルによる宣言

複数のモジュールで、同じ **TYPE...END TYPE, COMMON, DECLARE** ステートメントなどを使うときは、これらの宣言をインクルード ファイルにまとめて、それぞれのモジュールの先頭にインクルードすると便利です。インクルード ファイルでは、次のような項目をこの順序で指定します。

1. モジュール内の **SUB** または **FUNCTION** プロシージャで使うユーザー定義型変数パラメータに対する **TYPE...END TYPE** 定義
2. すべてのモジュールのプロシージャに対する **DECLARE** ステートメント
3. すべてのモジュールで共通に使う定数を定義する **CONST** ステートメント
4. そのモジュールと他のモジュールで共通に使う変数を宣言する **COMMON** ステートメント (**COMMON** の使い方については、「モジュール間での変数の共有」を参照してください)。

プロシージャの定義をまとめてあるモジュールを使うたびに、それらのプロシージャを呼び出す他のすべてのモジュールの先頭に **$INCLUDE**メタコマンドを挿入してください。通常は、プロシージャの定義を集めた各モジュールに対して 1 つのインクルード ファイルを作成し、そのモジュールを使うたびに対応するインクルード ファイルをインクルードするようにします。このようにすると、次のような利点があります。

- プロシージャの定義を集めたモジュールを、プログラムを作成するときの 1 つの部品として扱うことができます。つまり、別のモジュールに含まれるプロシージャを利用しようとするたびに、そのプロシージャに対する **DECLARE** ステートメントを 1 つ 1 つコピーする必要がありません。**$INCLUDE** メタコマンドを 1 つ挿入することによって、それらのプロシージャを使うことができます。
- プロシージャの宣言を含むインクルード ファイルを使うと、インクルード ファイルで定義されているプロシージャに対する **DECLARE** ステートメントの自動生成が行われません。
- このようなインクルード ファイルを使うと、別のモジュールで定義されている **FUNCTION** プロシージャを認識しないという問題を避けることができます (詳しくは、「DECLARE ステートメントが自動生成されない場合」を参照してください)。
- **COMMON** ブロックの不一致を避けることができます。

DECLARE ステートメントを自動生成する Visual Basic の機能を利用して、次の手順でインクルード ファイルを作成できます。

▼ インクルード ファイルで DECLARE ステートメントの自動生成機能を使用するには

1. モジュールを作成し、必要なプロシージャを定義します。
2. そのモジュールで定義されている、すべての **SUB** または **FUNCTION** プロシージャを呼び出します。

   実行開始モジュールでないコード モジュールでは、モジュール レベルに呼び出しを行うコードを仮に記述します。実行開始モジュールでなければ、通常の実行でモジュール レベルのコードが実行されることはありません。
3. モジュールを保管すると、すべてのプロシージャに対して **DECLARE** ステートメントが自動的に生成されます。
4. このモジュールを編集して、プロシージャ呼び出しを削除し、**DECLARE** ステートメントをインクルード ファイルに移動します。
5. **DECLARE** ステートメントに **AS ANY** が含まれているときは、適切なユーザー定義型の名前で置き換えます。

インクルード ファイルのテスト

Visual Basic では、インクルード ファイルの内容を調べる方法が用意されています。[表示] メニューには、[インクルード ファイル編集] コマンドと [インクルード ファイル表示] コマンドがあります。カーソルが **$INCLUDE** メタコマンドを含む行に置かれているときに、[インクルード ファイル表示] コマンドを選択すると、インクルード ファイルの内容がメタコマンドの下に表示されます。デフォルトの設定では、インクルード ファイル中のコードは、他のコードと別の色で表示されます。ただし、この状態ではインクルード ファイルの内容を編集することはできません。表示されているインクルード ファイル中のテキストを編集しようとすると、インクルード ファイルをロードするかどうかをたずねるダイアログ ボックスが表示されます。<確認> ボタンをクリックすると、インクルード ファイルの内容がロードされてコード ウィンドウに表示されます。また、[表示] メニューの [インクルード ファイル編集] コマンドを使うと、インクルード ファイルをロードして編集できます。さらに、[ファイル] メニューの [ファイル追加] コマンドでもインクルード ファイルをロードできますが、[ファイル追加] ダイアログ ボックスの <インクルード> オプション ボタンを選択しておかなければならないことに注意してください。このボタンを選択しないと、ファイルはコード モジュールと見なされます。インクルード ファイルをモジュールとしてロードしているときにアプリケーションを保管すると、インクルード ファイルの名前が .MAK ファイルに追加され、メモリを消費します。**$INCLUDE** メタコマンドの構文については、『リファレンス』またはオンライン ヘルプを参照してください。

重　要　インクルード ファイルは、アプリケーションの一部ですがモジュールではありません。**$INCLUDE** メタコマンドでファイル名を指定することによって、ファイルに含まれる情報がインクルードされます。インクルード ファイルの名前は、.MAK ファイルに追加しないように注意してください。

次の例で、コード モジュールとインクルード ファイルの使い方を示します。インクルード ファイルを作成するには、[ﾌｧｲﾙ] メニューの [新規モジュール] コマンドを選択して、ファイル名 SUPMOD.BI を入力し、<インクルード> オプションを選択します。

```
'=====================================================
'                        SUPMOD.BAS
'このモジュールでは, Prompter プロシージャと Max! プロシージャを定義
'します.
'=====================================================
'$INCLUDE: 'SUPMOD.BI'

STATIC FUNCTION Max! (X!, Y!)
    IF X! > Y! THEN Max! = X! ELSE Max! = Y!
END FUNCTION

STATIC SUB Prompter (Suffix$, RecVar AS RecType)
    Msg$ = "品名" + Suffix$
    RecVar.Description = INPUTBOX$(Msg$)
    Msg$ = "必要量" + Suffix$
    RecVar.Quantity = VAL(INPUTBOX$(Msg$))
END SUB

'=====================================================
'                        SUPMOD.BI
'このファイルは, Prompter プロシージャと Max! プロシージャの DECLARE
'ステートメントを含むインクルード ファイルです (ユーザー定義型 RecType を
'定義する TYPE ステートメントも含まれます). SUPMOD.BAS モジュールは,
'このファイルと組み合わせて使います.
'=====================================================
'
TYPE RecType
    Description AS STRING * 15
    Quantity AS INTEGER
END TYPE

DECLARE FUNCTION Max! (X!, Y!)
DECLARE SUB Prompter (Prefix AS STRING, RecVar AS RecType)

'=====================================================
'                        SAMPLE.BAS
'このモジュールはプログラムの実行開始モジュールで, SUPMOD.BAS モジュールを
'使っています.
'SUPMOD.BAS 中の Prompter プロシージャと Max! プロシージャを呼び出し
'ます.
'=====================================================
'
'SUPMOD.BI に含まれる宣言をインクルードします.
'$INCLUDE: 'SUPMOD.BI'
'
'
'
DIM MyRec AS RecType
A = VAL(INPUTBOX$("最初の値は: "))
B = VAL(INPUTBOX$("次の値は: "))
'SUPMOD.BAS の Max! FUNCTION プロシージャを呼び出します.
```

```
MSGBOX STR$(Max!(A, B))
'
'
'

'SUPMOD.BAS の Prompter SUB プロシージャを呼び出します.
Prompter "を入力してください:", MyRec
'
'
'
```

複数のモジュールで同じインクルード ファイルをインクルードしても、実行可能ファイルのサイズや実行速度には影響を与えません。ただし、プログラミング環境でアプリケーションを実行するときは、インクルード ファイルの参照を処理するために、実行速度が多少低下します。

Quick ライブラリのプロシージャの宣言

プロシージャを集めたコード モジュールは、Quick ライブラリにしておくと便利です。この場合も、Quick ライブラリに含まれるすべてのプロシージャに対する **DECLARE** ステートメントを 1 つのインクルード ファイルにまとめておけば、**$INCLUDE** メタコマンドを 1 行追加するだけで、Quick ライブラリ中のプロシージャを簡単に呼び出すことができます。

16.5　変数の共有

プロシージャは、引数リストとパラメータ リストによる変数の引き渡しの他に、次の 2 つの方法によって、他のプロシージャやモジュール レベル コード (モジュール内でプロシージャの外にあるコード) との間で変数を共有できます。また、複数のモジュールで変数を共有することもできます。ただし、フォーム モジュールのモジュール レベルには実行可能ステートメントを記述できませんので注意してください。

- プロシージャ内で **SHARED** ステートメントを使って変数を宣言すると、その変数は、そのプロシージャとモジュール レベル コードの間で共有されます。同じモジュール内の複数のプロシージャで共有したいときは、各プロシージャに **SHARED** ステートメントを記述します。
- モジュール レベルで、**COMMON SHARED**, **DIM SHARED**, **REDIM SHARED** ステートメントを使って変数を宣言すると、その変数は、モジュール レベル コードと、そのモジュール内のすべてのプロシージャの間で共有されます。この方法は、モジュール内のすべてのプロシージャで同じ変数を使いたいときに便利です。
- モジュール レベルで、**COMMON** ステートメントを使って変数を宣言すると、その変数を複数のモジュールで共有することができます。

ここでは、変数を共有するための 3 つの方法について順に説明して行きます。

モジュール内の特定のプロシージャ間での変数の共有

モジュール内の複数のプロシージャとモジュール レベル コードとの間で変数を共有したいときは、それぞれのプロシージャで **SHARED** ステートメントを使って変数を宣言します。**SHARED** ステートメントで配列変数を指定するときは、配列名の後ろに空のカッコを付けます。

```
DIM ArrayName(50)

SUB JustAnotherSub
    SHARED ArrayName ()
  .
  .
  .
```

その変数を宣言するときに **AS** キーワードを使って変数のデータ型を指定しているときは、**SHARED** ステートメントでも同じ指定を行わなければなりません。

```
DIM Buffer AS STRING * 10
CALL ReadRecords
  .
  .
  .
END

SUB ReadRecords ()
    SHARED Buffer AS STRING * 10
  .
  .
  .
END SUB
```

次の例では、GetRecords プロシージャと InventoryTotal プロシージャの間で、**SHARED** ステートメントを使って変数を共有しています。

```
DECLARE SUB GetRecords ()
DECLARE FUNCTION InventoryTotal! ()
'=========================================================
'                    モジュール レベル コード
'=========================================================
TYPE RecType
    Price AS CURRENCY
    Desc AS STRING * 35
END TYPE

DIM RecVar(1 TO 100) AS RecType          'ユーザー定義型の配列変数

FileSpec$ = INPUTBOX$ "ファイル名は: "
CALL GetRecords
PRINT InventoryTotal
END
```

```
'==========================================================
'                    プロシージャレベル コード
'==========================================================
STATIC SUB GetRecords ()

'FileSpec$ とユーザー定義型の配列変数 RecVar は，モジュール レベル
'コードとの間で共有されます．
    SHARED FileSpec$, RecVar() AS RecType
    OPEN FileSpec$ FOR RANDOM AS #1
  .
  .
  .
END SUB

STATIC FUNCTION InventoryTotal ()

'モジュール レベル コードとの間で共有されるのは、ユーザー定義型の配列変数
'RecVar だけです．
    SHARED RecVar() AS RecType
  .
  .
  .
END FUNCTION
```

モジュール内の全プロシージャでの変数の共有

モジュール レベルで変数を宣言する **COMMON, DIM, REDIM** ステートメントで **SHARED** キーワードを指定すると、そのモジュール内のすべてのプロシージャで変数を共有できます。つまり、**SHARED** キーワードを指定すると、変数はそのモジュールに対してグローバルになります。モジュール内のすべてのプロシージャで変数を共有したいとき、**SHARED** キーワードを使うと便利です。

次の例は、そのモジュールのすべてのプロシージャで共有される変数を宣言します。

```
COMMON SHARED A, B, C
DIM SHARED Array(1 TO 10, 1 TO 10) AS UserType
REDIM SHARED Alpha(N%)
```

次の例では、モジュール レベル コードと 2 つの **SUB** プロシージャ FillArray と PrintArray の間で、文字列配列 StrArray と整数変数 Min および Max を共有します。

```
'==========================================================
'                    モジュール レベル コード
'==========================================================

DECLARE SUB FillArray ()
DECLARE SUB PrintArray ()

'次の DIM ステートメントでは，整数変数 Min および Max と文字列配列
'StrArray を，このモジュールのすべての SUB または FUNCTION プロシージャ
'で共有することを宣言します．
DIM SHARED StrArray (33 TO 126) AS STRING * 5
DIM SHARED Min AS INTEGER, Max AS INTEGER
```

```
Min = LBOUND(StrArray)
Max = UBOUND(StrArray)

CALL FillArray          '変数は SHARED と指定されているので,
CALL PrintArray         '引数の受け渡しを行わないで済みます.

END

'==========================================================
'                    プロシージャレベル コード
'==========================================================

STATIC SUB FillArray
    '配列の 33 番目から 126 番目までの各要素に 5 文字の文字列を格納します.
    'それぞれの文字は ASCII コード I% を表します.
    FOR I% = Min TO Max
        StrArray(I%) = STRING$(5, I%)
    NEXT I%
END SUB

STATIC SUB PrintArray
    FOR I% = Min TO Max
        PRINT StrArray(I%)
    NEXT I%
END SUB
```

出力例

```
!!!!!
"""""
#####
$$$$$
%%%%%
&&&&&
'''''
  .
  .
  .
```

プログラミング環境以外で、市販のテキスト エディタなどで編集したプログラムをコンパイルするときは、**SHARED** キーワードを指定した変数の宣言がプロシージャ宣言より前になければならないことに注意してください。変数宣言がプロシージャ宣言よりも前にないと、プロシージャ内でこれらの変数を共有することができません (Visual Basic を使ってプログラムを作成するときは、プログラムは自動的に正しい順序で保管されるので、このような注意は必要はありません)。

```
STATIC FUNCTION Adder (X, Y)
    Adder = X + Y + Z
END FUNCTION

DIM SHARED Z
Z = 2
MSGBOX STR$(Adder (1, 3))
END
```

このプログラムを実行すると、**MSGBOX** ダイアログ ボックスに次のように表示されます。

```
4
```

このモジュールを保管するときは、次のように、DIM SHARED Z ステートメントの後ろで Adder を定義しなければなりません。

```
DECLARE FUNCTION Adder (X, Y)

'変数 Z は Adder との間で共有されます.
DIM SHARED Z
Z = 2
MSGBOX Str$(Adder (1, 3))
END

STATIC FUNCTION Adder (X, Y)
    Adder = X + Y + Z
END FUNCTION
```

このプログラムを実行すると、**MSGBOX** ダイアログ ボックスに次のように出力されます。

```
6
```

変数宣言の優先順位

参照によって引き渡された変数を除き、特に共有を宣言していなければ、プロシージャ内で使われる変数は、そのプロシージャに対してローカルです。**SHARED** ステートメントで宣言すると、プロシージャとモジュール レベルで変数を共有できます。この場合、プロシージャとモジュール レベルでは、同じ変数名を使います。

モジュール レベルで **SHARED** キーワードを付けて宣言された変数は、そのモジュールに対してグローバルになります。この場合、プロシージャ内で特に宣言はしません。**SHARED** キーワード付きですでに宣言されている変数名を使えば、モジュール レベルと同じ変数が参照されます。ただし、プロシージャ内で **DIM** ステートメントを使って再宣言すると、モジュール レベルで **SHARED** キーワードを付けて宣言された変数と同じ名前であっても、プロシージャ内でローカルな変数として扱うことができます。また、プロシージャ内で **STATIC** ステートメントで宣言した変数も、グローバルな変数や定数と同じ名前であっても、そのプロシージャに対してローカルな変数になります。

また、フォームを使うプログラムでは、プロパティ名と同じ名前の変数を使わないように注意してください。プロパティ名は予約語ではないため、変数や定数の名前として宣言できます。例えばコントロール配列を使うアプリケーションでは、イベントが発生したコントロールのインデックス番号 (Index プロパティの値) が渡されるパラメータとして、デフォルトで Index という名前が使われます。この場合、モジュール レベルで **SHARED** キーワードを付けて Index という名前の変数を宣言していても、このイベント プロシージャ内では無効になります。

モジュール間での変数の共有

複数のモジュールで変数を共有するときは、各モジュールのモジュール レベルで **COMMON** または **COMMON SHARED** ステートメントを使って変数を宣言します。

次の例では、モジュール間で変数を共有するために、**SUB** プロシージャを呼び出すモジュールで **COMMON** ステートメントを使って変数を宣言し、プロシージャを定義するモジュールで **COMMON SHARED** ステートメントを使って変数を宣言しています。**COMMON SHARED** が指定されているので、このモジュールのすべてのプロシージャが共有変数にアクセスできます。

```
'==========================================================
'                        メイン モジュール
'==========================================================

COMMON A!, B!
A! = 2.5
B! = 1.2
CALL Square
CALL Cube
END

'==========================================================
'          Cube および Square プロシージャを含むモジュール
'==========================================================

'注意: 変数 (X!, Y!) の名前は, 他のモジュールにおける変数名 (A!, B!) と
'同じである必要はありませんが, データ型は同じでなければなりません.

COMMON SHARED X!, Y!      'このステートメントはモジュール レベルにあり
                          'ます. 変数 X! と Y! は, Cube および Square
                          'プロシージャで共有されます.
STATIC SUB Cube
    PRINT "A! の 3 乗 ="; X! ^ 3
    PRINT "B! の 3 乗 ="; Y! ^ 3
END SUB

STATIC SUB Square
    PRINT "A! の 2 乗 ="; X! ^ 2
    PRINT "B! の 2 乗 ="; Y! ^ 2
END SUB
```

Visual Basic では、プログラムは多くの場合に複数のモジュールから構成されます。名前付きの **COMMON** ブロックを使うと、変数をグループ分けして、それぞれのモジュールで必要な変数だけを共有できます。次の例では、モジュール レベルで名前付きの **COMMON** ブロックを使い、それぞれのプロシージャで異なる変数のグループを共有しています。

```
DECLARE SUB VolumeCalc ()
DECLARE SUB DensityCalc ()
```

```
'==========================================================
'                    メイン モジュール
'入力された値から円柱の体積と密度を求めて表示します.
'==========================================================

COMMON /V/ Volume
COMMON /HR/ Height, Radius
COMMON /WD/ Weight, Density

Height = VAL(INPUTBOX$ ("円柱の高さ (センチメートル) : "))
Radius = VAL(INPUTBOX$ ("円柱の半径 (センチメートル) : "))
Weight = VAL(INPUTBOX$ ("円柱の重さ (グラム) : "))

CALL VolumeCalc
CALL DensityCalc

Msg$ = "体積は  " + STR$(Volume) + " 立方センチメートルです."
MSGBOX Msg$
Msg$ = "密度は  " + STR$(Density) + " グラム/立方センチメートルです."
MSGBOX Msg$
END

'==========================================================
'   DensityCalc プロシージャを定義するモジュール
'==========================================================

COMMON /V/ Vo
COMMON /WD/ We, De

STATIC SUB DensityCalc
    'このプロシージャでは, 変数 Weight, Volume, Density を共有します.
    SHARED We, Vo, De
    De = We / Vo
END SUB

'==========================================================
'   VolumeCalc プロシージャを定義するモジュール
'==========================================================

COMMON /V/ Vo
COMMON /HR/ He, Ra

STATIC SUB VolumeCalc
    'このプロシージャでは, 変数 Height, Radius, Volume を共有します.
    SHARED He, Ra, Vo
    CONST PI = 3.141592653589#
    Vo = PI * He * (Ra ^ 2)
END SUB
```

このプログラムを実行すると、**INPUTBOX** および **MSGBOX** ダイアログ ボックスに、次のように表示されます。ユーザーが値を入力すると、円柱の体積と密度が表示されます。

円柱の高さ（センチメートル）：100

円柱の半径（センチメートル）：10

円柱の重さ（グラム）：10000

体積は 31415.93 立方センチメートルです.

密度は .3183099 グラム/立方センチメートルです.

他のモジュールとのプロパティの共有

$FORM メタコマンドを使うと、あるフォーム モジュールまたはコード モジュールから、別のフォーム モジュール中のオブジェクトのプロパティを参照できます。**$FORM** メタコマンドは、フォームのプロパティを操作するモジュールのモジュール レベルに指定します。他の宣言と同じように、インクルード ファイルに **$FORM** メタコマンドを記述することもできます。

DECLARE ステートメントと同じように、プログラミング環境でプログラムを保管すると、必要に応じて **$FORM** メタコマンドが自動生成されます。

変数の重複

多くの変数やプロシージャを含む長いプログラムでは、“変数の重複”という問題が起こることがあります。変数の重複は、次のような場合に起こります。

- プロシージャに引き渡される引数リストに、同じ変数が 2 回以上現れる場合
- 引数リストで引き渡される変数を、**SHARED** ステートメントまたは **SHARED** キーワードを付けて宣言している場合

変数の重複が起きないように、プロシージャで共有されている変数が、プロシージャを呼び出すときに使う引数リストに現れていないか、十分にチェックしてください。また、同じ名前の変数を 2 回引き渡さないように注意してください。

```
'Test プロシージャには X が 2 回引き渡されているので，変数の重複が起きます.
CALL Test(X, X, Y)
```

次の例は、どのような場合に変数の重複が起きるかを示します。このプログラムでは、**DIM SHARED** ステートメントを使って、モジュール レベル コードと **SUB** プロシージャの間で変数 A を共有しています。ここで、A が参照によって **SUB** プロシージャに引き渡されているため、プロシージャ内では、A と X がメモリ内の同じ位置を参照しています。**SUB** が X の値を変更すると、A の値も変更されます。また、A の値を変更すると、X の値も変更されます。

```
DIM SHARED A
A = 4
CALL PrintHalf(A)
END
```

```
STATIC SUB PrintHalf (X)
    Msg$ = STR$(X) + " ÷ 2 に " + STR$(A) + " ÷ 2 を加えると "
    X = X / 2        'X と A は 2 になります.
    A = A / 2        'X と A は 1 になります.
    Msg$ = Msg$ + STR$(A + X) + " になります"
    MSGBOX$ Msg$
END SUB
```

このプログラムを実行すると、**MSGBOX** ダイアログ ボックスに次のように表示されます。

4 ÷ 2 に 4 ÷ 2 を加えると 2 になります

16.6　自動変数と STATIC 変数

SUB および **FUNCTION** ステートメントで **STATIC** キーワードを指定すると、プロシージャ内のローカル変数は静的 (STATIC) 変数として扱われます。このときローカル変数の値は、次にプロシージャが呼び出されるまで保持されます。一般に、プロシージャの変数をすべて静的 (STATIC) 変数とすると、プロシージャの実行速度が多少上がります。

STATIC キーワードを指定しないと、プロシージャ内のローカル変数はデフォルトで自動変数になります。このとき、ローカル変数の値は、プロシージャが呼び出されるたびに、数値変数は 0 に、また文字列変数は null 文字列 ("") に初期化されます。

ローカル変数の値の保存

プロシージャ内で **STATIC** ステートメントを使うと、指定した変数だけを静的 (STATIC) 変数にすることができます。また、**STATIC** ステートメントで指定された変数は、自動的にローカル変数になります。

STATIC ステートメントは、プロシージャの中にしか記述できません。**STATIC** ステートメントで配列を指定するときは、配列名の後ろに空のカッコを付けます。また、次の例に示すように、**STATIC** ステートメントで配列を指定したときは、その配列を使う前に配列のサイズを宣言しておかなければなりません。ただし、2 回目に呼ばれたときには、宣言しないでください。

```
SUB SubProg2 ()
    STATIC Array() AS INTEGER
    DIM Array(-5 TO 5, 1 TO 25) AS INTEGER
  .
  .
  .
END SUB
```

STATIC ステートメントで **AS** キーワードを使うときは、**STATIC** ステートメントと **DIM** (または **REDIM**) ステートメントの両方で **AS** キーワードを使わなければなりません。

次の例は、**STATIC** ステートメントで指定されている文字列変数 **Y$** の値が **TestSub** を続けて呼び出す間、保存されることを示しています。

```
DECLARE SUB TestSub ()
FOR I% = 1 TO 5
    TestSub          'TestSub を 5 回呼び出します.
NEXT I%
END

SUB TestSub          '注意: STATIC キーワードは指定されていません.
'TestSub 内では X$ と Y$ はローカル変数です (すなわち, これらの変数の値は
'モジュール レベルとの間で共有されません. X$ は自動変数なので,TestSub を
'呼び出すたびに null 文字列に再初期化されます. しかし, Y$ は静的変数なので,
'最後に呼び出したときの値が保存されます.
    STATIC Y$
    X$ = X$ + "*"
    Y$ = Y$ + "*"
    PRINT X$, Y$
END SUB
```

このプログラムを実行すると、次のように表示されます。

```
*        *
*        **
*        ***
*        ****
*        *****
```

16.7　再帰プロシージャ

Visual Basic のプロシージャは、再帰プロシージャとすることができます。再帰プロシージャとは、自分自身を呼び出したり、呼び出した他のプロシージャから再び自分自身が呼び出されるようなプロシージャをいいます。

階乗の計算

再帰プロシージャを説明するときは、数学の階乗関数がよく使われます。n! (n の階乗) は、次の方程式で定義されます。

n! = n * (n-1) * (n-2) * ... * 2 * 1

例えば、5 の階乗は次のように求めます。

5! = 5 * 4 * 3 * 2 * 1 = 120

注　意　数学の階乗記号 (!) を Visual Basic の単精度数の型宣言文字と混同しないでください。

階乗は、再帰的に定義することもできます。

n! = n * (n-1)!

例えば、次のようになります。

5! = 5 * 4!
4! = 4 * 3!
3! = 3 * 2!
2! = 2 * 1!
1! = 1 * 0!

再帰を定義するときは、終了条件を指定しなければなりません。階乗の計算では、0! を求めるときが終了条件となります。0! は 1 です。

注　意　再帰プロシージャでは、**STATIC** キーワードを指定して、デフォルトで静的 (STATIC) 変数にすることができます (次の例を参照)。ただし、デフォルトで自動変数にした方がよい場合もあります。自動変数を使うと、再帰呼び出しによって変数の値が変化することはありません。

次の例では、再帰 **FUNCTION** プロシージャを使って、階乗を計算します。

```
DECLARE FUNCTION Fact# (N%)
DO
    Num% = VAL(INPUTBOX$ ("0-20 の値 (-1 で終了): "))
    IF Num% >= 0 AND Num% <= 20 THEN
        Msg$ = STR$(Num%) + "の階乗は" + STR$(Fact#(Num%))
        MsgBox Msg$
    END IF
LOOP WHILE Num% >= 0
END

STATIC FUNCTION Fact# (N%)
    'N が 0 よりも大きいときは, 再度 Fact# を呼び出します.
    IF N% > 0 THEN
        Fact# = N% * Fact#(N% - 1)
    ELSE                    '再帰呼び出しの終了条件 (N% = 0) になったので,
                            'それぞれの再帰から戻ります.
        Fact# = 1
    END IF
END FUNCTION
```

再帰の実用的な使い方

階乗関数は、再帰の考え方を説明するための簡単な方法ですが、数学計算用のプログラム以外では、あまり役に立ちません。ここでは、再帰の実用的な使い方として、ディスク ドライブのディレクトリ構造など、ツリー構造を再帰的に検索するプログラムを示します。サンプルの SEEK アプリケーションでは、ディレクトリの構造を再帰的に検索しています。DD 関数を呼び出すときは、3 つの引数を引き渡します。最初の引数は再帰を開始する現在のパス、2 番目の引数はこのパスのディレクトリの数、3 番目の引数は制御が戻るときのパスをそれぞれ指定します。**FUNCTION** の行には **STATIC** キーワードが指定されていないので、プロシージャ中の変数は、デフォルトで自動変数になります。DD 関数は、ツリーの各ノードを最後まで検索し、それぞれの再帰から

戻りながら、検索パターンに一致するファイル名のパス指定をリスト ボックスに表示します。

```
FUNCTION DD%(NewPath$, DirCount%, BackUp$)
  .
  .
  .
    DirsToPeek% = Dir1.ListCount     'サブディレクトリの数
    DO WHILE DirsToPeek% > 0
        OldP$ = Dir1.Path            '先頭のパス指定を保存します.
        Dir1.Path = NewPath$         'Dir1 リスト ボックスをリセット
                                     'します.
        IF Dir1.Liscount > 0 THEN    'まだディレクトリがあるとき
            Dir1.Path = Dir1.List(DirsToPeek% -1)
            result% = DD%((Dir1.Path), DirCount% -1, OldP$)
        END IF
        DirsToPeek% = DirsToPeek% - 1
                                 'ディレクトリ内を 1 レベル移動します.
    LOOP
    '終了する前に, このディレクトリのファイルをリスト ボックスに表示します.
    IF File1.Listcount THEN
        IF LEN(Dir1.Path) <= 3 THEN
            ThePath$ = Dir1.Path
        ELSE
            ThePath$ = ThePath$ + "\"
        END IF
        FOR ind% = 0 to (File1.ListCount -1)
            entry$ = The paths$ + File1.List(ind%)
            1stFoundFiles.ADDITEM entry$
              .
              .
              .
        NEXT ind%
    END IF
    '先頭に戻らないときは, Dir1 リスト ボックスをリセットします.
    IF BackUp$ THEN Dir1.Path = BackUp$
EXIT FUNCTION

'エラー処理ルーチンをここに記述します.

END FUNCTION
```

この例で、再帰プロシージャを呼び出すとき、最初の引数 Dir1.Path をカッコで囲んで引き渡していることに注意してください。これにより、関数のパラメータ NewPath$ の現在のパスが、テンポラリ変数に格納されます。

スタックサイズの調整

再帰を使うと、メモリを大量に消費します。これは、再帰呼び出しから制御が戻ったときに、呼び出し前の正しい変数の値が得られるように、**SUB** または **FUNCTION** プロシージャ内の自動変数が、呼び出しのたびにスタック上に保存されるからです。

スタック サイズが不足すると、“スタック領域が不足です” エラーが発生します。このエラーは修復できないランタイム エラーなので、起きないような対策が必要です。

多くの自動変数を含む再帰プロシージャや、DD プロシージャのように再帰プロシージャのネストが深いときは、プロシージャを実行する前にスタックサイズを調整します。

スタックサイズを調整するには、**FRE** および **STACK** 関数と **STACK** ステートメントを使います。

スタックサイズを再設定する前に、いくつか調べなければならない項目があります。まず、再帰プロシージャが必要とするメモリ量を調べます。

▼ **プロシージャが必要とするスタック領域を算出するには**

1. プロシージャの **DECLARE** ステートメント，プロシージャを呼び出すステートメント，プロシージャ自身を含むテスト モジュールを作成します。
2. 再帰プロシージャを呼び出す直前と直後に、未使用のスタック領域のサイズを返す関数 FRE(-2) を追加します。この関数の戻り値は、2 つとも長整数として格納するようにしてください。
3. テスト モジュールを実行します。2 つの **FRE** 関数の戻り値の差が、プロシージャの 1 回の呼び出しで使われるスタック領域のサイズ (バイト単位) になります。
4. プロシージャが呼び出される最大回数を予測して、1 回の呼び出しで使われるスタック サイズにこの値をかけます。この結果が、再帰プロシージャで必要なメモリ サイズになります。

プロシージャが必要とするスタック領域のバイト数を求めたら、現在割り当てられているスタック サイズを調べます。スタック サイズのデフォルトは、2.5K バイト～ 3.5K バイトです。ただしこのサイズは、プログラムがコンパイルされているかプログラミング環境で実行しているかなど、状況によって変わります。スタック サイズは、**STACK** ステートメントを使って変更できます。デフォルトのスタック サイズを使っているときは、次のプログラムでスタック サイズを調整できます (必要なスタック領域を割り当てることができるとき)。

```
'現在割り当てられているスタック サイズを含む変数を初期化します.
CurrentSize = 3072
'再帰プロシージャに必要なスタック サイズを説明した方法で求め，その値で
'変数を初期化します.
RecursiveBytes = 6000
'現在使われているスタック サイズを調べます.
BytesOnStack = CurrentSize - FRE(-2)
'必要なスタック領域の合計を求めます.
RequiredSpace = RecursiveBytes + BytesOnStack
'必要なスタック領域を割り当てることができるときは，スタック サイズを変更
'します.
IF RequiredSpace <= STACK THEN
    STACK RequiredSpace
ELSE
    GOTO ReportError
END IF
```

STACK ステートメントは、モジュール レベルでのみ使います。フォームを使うアプリケーションでスタック サイズを設定するには、コード モジュールでアプリケーションの実行を開始し、**LOAD** ステートメントまたは **SHOW** メソッドを使ってアプリケーションの最初のフォームをロードしなければなりません。

この例では、**STACK** ステートメントと **STACK** 関数を使っていることに注意してください。**STACK** 関数は、スタックに割り当て可能な最大値を返します。また、**STACK** ステートメントは、実際にスタック領域を割り当てます。詳しくは、「リファレンス」またはオンライン ヘルプを参照してください。

17 アプリケーションのデバッグ

この章では、Visual Basic for MS-DOS のデバッグ機能を使って、アプリケーションの動作を分析し、プログラムの誤動作 (バグ) の原因を調べる方法について説明します。また、アプリケーションのバグを最小限に押えるためのプログラミング テクニックと設計上の留意点についても説明します。

Visual Basic for MS-DOS のプログラミング環境は、QuickBasic や Microsoft Basic Professional Development System でも用意されていた、次のようなデバッグ機能をすべてサポートしています。

- ブレークポイントやウォッチ ポイントの設定
- シングル ステップ実行，プロシージャ ステップ実行
- 変数やプロパティの値を監視するためのデバッグ ウィンドウ (従来のウォッチ ウィンドウ)
- プロシージャが呼び出された経過を表示するコール ヒストリ ウィンドウ
- ヒストリの記録
- 編集後の継続実行
- 次に実行するステートメントの設定
- 実行が中断しているときのプロシージャのテスト

17.1 アプリケーションのエラー

アプリケーションの作成時や実行時に発生するエラーは、大きく次の 3 種類に分類できます。

- **構文エラー**

  主にアプリケーションの作成中に発生します。入力したステートメントが構文に反しているときや、指定しなければならない項目が指定されていないときなどに発生します。

- **ランタイム エラー**

  アプリケーションの実行中に発生します。**KILL** ステートメントで存在しないファイルを削除しようとするなど、実行不可能な操作を行うと発生します。

■ **論理エラー**

アプリケーションの実行中に発生します。プログラムの実行は正常終了しても、プログラムの論理に問題があり、期待した結果が得られない場合を指します。

構文エラー

構文エラーは、不正なステートメントを入力したり、必要な項目が入力されていないときに発生します。単なるキーワードのタイプミスや、ステートメントに必要な記号や引数が指定されていない場合も構文エラーになります。構文エラーは、[ENTER] キーを押すなど、エラーとなる行からカーソルを動かすとすぐに検出されます。

構文エラーの中には、プログラムを実行したときや、コンパイルして実行可能ファイルを作成したときに検出されるものもあります。例えば、引数の数や型をまちがえて指定したままプロシージャを呼び出していても、プログラムを実行するかコンパイルするまで、この構文エラーは検出されません。また、**IF...END IF** ステートメントや **FOR...NEXT** ステートメントでキーワードの対応が正しくないときも、プログラムを実行するまで構文エラーは検出されません。

ブロック構造やループ構造がネストしていると、実際の原因とは異なった構文エラーが検出される場合があります。例えば、次のプログラムを実行すると、“IF ブロックがありません” というエラー メッセージが出力されます。

```
IF MyVar <= YourVar THEN
    FOR counter = startVal TO EndVal
    CALL ValCompare
END IF
```

しかし、このプログラムの **IF...END IF** ブロックは正しく記述されています。実際には **FOR...NEXT** ループの **NEXT** ステートメントが抜けているために構文エラーが発生しているわけですが、**FOR...NEXT** ループが **IF...END IF** ブロック内でネストしているため、**END IF** ステートメントが無効なステートメントと見なされ、エラーとして検出されています。このプログラムはエラーが修正されるまでコンパイルも実行もできません。

ランタイム エラー

ランタイム エラーは、アプリケーションの実行時、ステートメントが実行できない操作や演算を行ったときに発生します。アプリケーションを実行しない限り検出されないため、いつどこで発生するかを予測するのは困難です。

よく見られる例は、0 による除算です。

```
FUNCTION Speed (miles AS SINGLE, hours AS SINGLE) AS SINGLE
    Speed = miles / hours        '時速を計算します。
END FUNCTION
```

この例のステートメントは構文的には正しく、構文エラーにはなりません。しかし、hours が 0 になると、この除算は実行できない演算となり、“0 で除算しました” というエラー メッセージが表示されます。

アプリケーションを実行するとき、ある状況になった場合だけに発生するランタイムエラーもあります。例えば、データをファイルに書き込もうとしたときに、ディスクに十分な空き容量がなかった場合は、“ディスクの空き容量が不足です”エラーが発生します。このようなエラーは、あらかじめエラー処理ルーチンを用意することで回避できます。エラー処理ルーチンについて詳しくは、第9章「エラー ハンドリング」を参照してください。

論理エラー

論理エラーは、プログラムの実行は正常に行われるが、結果が意図したものにならないときに発生します。プログラムには構文的な問題はなく、ランタイム エラーも発生しないので、なぜまちがった結果になるのかはプログラムの動きを調べなければ確認できません。

論理エラーが発生する原因にはさまざまな場合が考えられますが、変数の初期化を忘れたり、演算子の使い方やプロシージャの呼び出し方がまちがっていることがほとんどです。

ステートメントの動作をよく理解していないために発生する論理エラーもあります。例えば、次の例のような範囲を **SELECT CASE** ステートメントに指定してしまうことがあります。

```
SELECT CASE CurrentVal
    CASE -1 TO -5
    CalcValue(CurrentVal)
  .
  .
  .
```

2行目の **CASE** キーワード以降は構文に従って記述されており、構文エラーやランタイム エラーは発生しません。しかし、**TO** キーワードで範囲を表すときは、小さい値を先に指定しなければならないため、このままでは **CASE** キーワードに続くステートメント ブロックが実行されることはありません。正しく実行させるためには、範囲の指定を -5 TO -1 とします。

同じような論理エラーは、**FOR...NEXT** ループでも発生することがあります。

```
FOR counter = -1 TO -5
  .
  .
  .
NEXT counter
```

この例の **FOR** ステートメントも、構文に従って記述されているため、構文エラーは発生しません。しかし、**FOR...NEXT** ステートメントでループが実行されるたびに加算される値のデフォルトは1なので、このループ内のステートメントは実行されません。ループを正しく動作させるためには、**STEP** キーワードを使い、次のように記述します。

```
FOR counter = -1 TO -5 STEP -1
```

このように論理エラーは、構文エラーやランタイム エラーと比べて発見が困難と言えます。そこで Visual Basic のデバッグ機能を利用すると、効率よく論理エラーの発見と修正を行うことができます。

17.2　デバッグ機能

Visual Basic のデバッグ機能の多くは、[デバッグ] メニューと [実行] メニューから選択します。また、シングル ステップ実行 ([F8])，プロシージャ ステップ実行 ([F10])，ヒストリのトレース バック ([SHIFT+F8]) のように、ショートカット キーとしてファンクション キーが利用できるものもあります。

コール ヒストリ ウィンドウ，デバッグ ウィンドウ，イミディエイト ウィンドウも、デバッグを行う場合にたいへん役に立ちます。これらのウィンドウについては、オンライン ヘルプに詳しく説明されています。

ここでは、デバッグ機能の使い方を説明します。

[実行] メニュー	ショートカット キー	目的と特徴
[スタート]	[SHIFT]+[F5]	変数をすべて初期化し、プログラムを先頭から実行します。
[リスタート]		変数をすべて初期化してプログラムの実行を開始しますが、最初の実行可能ステートメントで実行を中断します。
[継続]	[F5]	中断されたプログラムの実行を、カレント ステートメントから継続します。
[COMMAND$ 編集]		コマンド ライン引数をシミュレートします。アプリケーションがコマンドライン引数を受け取ることができるときに使います。

D/ﾃﾞﾊﾞｯｸﾞ

```
A/ウォッチ追加...
I/インスタント ウォッチ... SHIFT+F9
W/ウォッチ ポイント...
D/ウォッチ解除...
L/全ウォッチ解除

T/トレース ON/OFF
H/ヒストリ ON/OFF

B/ブレークポイント ON/OFF        F9
C/全ブレークポイント解除
E/エラー時中断 ON/OFF
S/次のステートメント設定
```

[ﾃﾞﾊﾞｯｸﾞ] メニュー	ショートカットキー	目的と特徴
[ウォッチ追加]		変数の値をデバッグ ウィンドウで監視します。
[インスタント ウォッチ]	[SHIFT]+[F9]	変数の値を調べますが、デバッグ ウィンドウには表示しません。
[ウォッチ ポイント]		実行を中断する変数の条件を設定します。
[ウォッチ解除]		デバッグ ウィンドウから項目を 1 つ削除します。
[全ウォッチ解除]		デバッグ ウィンドウからすべての項目を削除します。
[トレース ON/OFF]		コード ウィンドウで、実行中のステートメントを次々に表示しながらコードを実行するかどうかを切り換えます。
[ヒストリ ON/OFF]		最後に実行された 20 行分のステートメントを記録するかどうかを切り換えます。
[ブレークポイント ON/OFF]	[F9]	現在行に対してブレークポイントを設定または解除します。
[全ブレークポイント解除]		すべてのモジュールのすべてのブレークポイントを解除します。
[エラー時中断 ON/OFF]		実行されたエラー処理ルーチンの最初のステートメントで実行を中断するかどうかを切り換えます。
[次のステートメント設定]		次にステップ実行または実行するステートメントを指定します。

メニューに含まれていない機能	ショートカットキー	目的と特徴
OPTION EXPLICIT		変数を宣言せずに参照するとエラーが発生するようにします。
出力画面	[F4]	アプリケーションの出力画面を表示します。
カーソル行まで実行	[F7]	カーソルが現在実行されている行にあるときは、[F7] を押すとその行を実行します。その他の場合は、[継続] コマンドと同じように動作します。
シングル ステップ実行	[F8]	カレント ステートメントを実行します。そのステートメントがプロシージャ呼び出しのときは、プロシージャが呼び出され、最初のステートメントを実行します。
プロシージャ ステップ実行	[F10]	カレント ステートメントを実行します。そのステートメントがプロシージャ呼び出しのときは、プロシージャ全体を 1 つのステートメントのように扱って実行します。
ヒストリのトレース	[SHIFT]+[F8]	プログラムが中断状態のとき、カレント ステートメントを 1 ステートメント分だけ戻します。

W/ｳｨﾝﾄﾞｳ

N/新規ウィンドウ
A/再配置

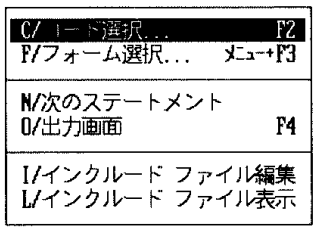
C/コール ヒストリ
D/デバッグ
H/ヘルプ
I/イミディエイト
O/出力画面
P/プロジェクト

*1 Untitled

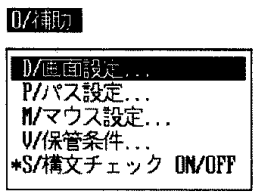
[ｳｨﾝﾄﾞｳ] メニュー	ショートカットキー	目的と特徴
[コール ヒストリ]		これまでにプロシージャが呼び出された経過を、最も新しく呼び出されたものから順に表示します。表示だけが可能ですが、一覧表示されたプロシージャ中を移動できます。
[デバッグ]		ウォッチしている式とウォッチ ポイントの状態を表示します。表示，コピー，印刷が可能です。

[ヘルプ]	[F1]	ヘルプを表示します。表示，コピー，印刷が可能です。
[イミディエイト]		実行を中断しているプログラムに含まれる変数を操作します。表示，書き込み，編集，実行，コピー，印刷が可能です。
[出力画面]	[F4]	画面出力を表示します。表示だけが可能です。
[プロジェクト]		プロジェクト (アプリケーション) に含まれるすべてのフォームおよびコード モジュールを表示します。

V/表示



[表示] メニュー	ショートカット キー	目的と特徴
[コード選択]	[F2]	モジュールとプロシージャを一覧表示します。コード ウィンドウで編集する内容を切り換えるために使います。
[フォーム選択]	[メニュー]+[F3]	フォームを一覧表示します。また、プログラミング環境とフォーム デザイナの切り換えのために使います。
[次のステートメント]		カーソル位置の行を、次に実行するステートメントとして設定します。
[出力画面]	[F4]	アプリケーションの出力画面を表示します。
[インクルード ファイル編集]		カーソル位置の行で指定されているインクルード ファイルを、編集できる状態でコード ウィンドウに表示します。
[インクルード ファイル表示]		現在編集中のプロジェクトに含まれるインクルード ファイルを、読み込み専用テキストとしてコード中に表示します。



[補助] メニュー	ショートカット キー	目的と特徴
[構文チェック ON/OFF]		ステートメントの指定が不適切なとき、エラー メッセージを表示するかどうかを切り換えます。

17.3　ヘルプの使い方

ヘルプは、プログラミング環境やフォーム デザイナの中からいつでも呼び出して参照できます。コードを記述しているときやデバッグ中でも参照できるので、構文エラーやランタイム エラー，定数の値，変数やプロシージャの使い方など、知りたい情報を状況に応じて表示することができます。さまざまな場面で、ヘルプを表示したいときは [F1] キーを押してみてください。

エラー時のヘルプの表示

構文エラーが発生したときは、コード ウィンドウに表示されているステートメントの名前やそこで使われているキーワードにカーソルを合わせて [F1] キーを押せば、そのキーワードを説明したヘルプ トピックが表示されます。

ランタイム エラーが発生すると、エラー メッセージを含むダイアログ ボックスが表示されます。ダイアログ ボックスが表示されているときに [F1] キーを押すと、そのランタイム エラーに関するヘルプ トピックが表示されます。

変数の値とプロシージャの値の表示

コード ウィンドウに表示されている変数名にカーソルを合わせて [F1] キーを押すと、その変数が使われているすべてのモジュールとプロシージャが一覧表示されます。この一覧には、変数のデータ型も表示されます。

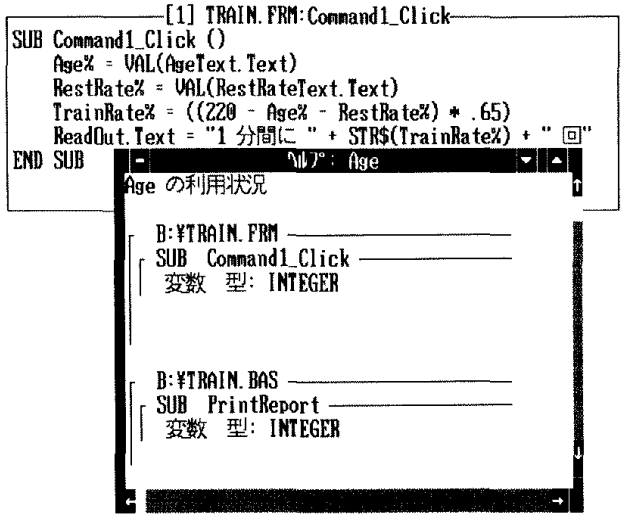


図 17.1　変数が使用されている場所のヘルプ表示

定数の値の表示

ヘルプでは、変数の現在の値を表示することはできませんが、定数の値は表示できます。数値を直接入力する代わりに記号定数を使い、デバッグの効率を上げることができます。次の例では、ランタイム エラー コードを記号定数として宣言しています。ランタイム エラーのコードとメッセージの一覧は、『リファレンス』の付録 D を参照してください。

```
CONST Err_InvalidFileName = 64
   .
   .
   .
SELECT CASE ERR
    CASE ERR_InvalidFileName
    Msg$ = "ファイルの指定がまちがえています。"
```

記号定数の現在の値を知りたいときは、その定数の上にカーソルを置いて [F1] キーを押します。その定数のデータ型と値、その定数が定義されているモジュール名 (プロシージャ レベルの定数のときはさらにプロシージャ名) が表示されます。

17.4　アプリケーションの設計

Visual Basic のデバッグ機能を利用すると、アプリケーションを実行させながら、その状態の監視，修正，テストを行えるため、アプリケーションのパフォーマンスや効率を向上させるのに役立ちます。例えば、アプリケーションの動作が予想よりも遅いときは、コードを 1 ステートメントずつ実行してみてください。適切ではないアルゴリズムや不要なコードを発見できることがあります。

アプリケーションの開発は、一般的に次の手順で行います。

1. はじめは、エラー処理や実行効率についてはあまり気にせずに、まずアプリケーションを作成します。
2. いろいろなデータを与えてテストし、バグを取ります。
3. パフォーマンス (サイズまたは実行速度) を向上させる方法を考えます。

Visual Basic には、アプリケーション開発の各段階で使用できるデバッグ機能が用意されています。しかし、標準的な設計やデバッグのテクニックを覚えておくと、デバッグ機能の利用をできるだけ減らすことができます。

設計のテクニックとデバッグ テクニック

デバッグの基本は、まずアプリケーションの動作中に何が起こっているのかを理解することです。アプリケーションの動作が理解できていれば、バグをより早く見つけることができます。次のことに注意してください。

- 関連するイベントと、それぞれのイベントに応答するプログラムを書き出して、アプリケーションを設計します。イベント プロシージャとジェネラル プロシージャは、それぞれの役割を明確にして作成します。それぞれのプロシージャがコンパクトで簡潔明瞭であれば、デバッグは簡単になります。
- **OPTION EXPLICIT** ステートメントを使います。このステートメントをコードのモジュール レベルに記述しておくと、変数を宣言せずにその変数を参照すると、エラーが発生します。これにより、変数名のスペル ミスという、単純でかつ発見しにくいバグを防ぐことができます。
- コメントを詳細に付けます。プログラムの各ブロックの役割をコメントとして記述しておけば、あとで見直して分析するときに理解しやすくなります。
- フォームに配置したコントロールの動作を理解します。チェック ボックスのように動作がわかりやすいコントロールもありますが、ドライブ リスト ボックス，ディレクトリ リスト ボックス，ファイル リスト ボックスの各コントロールなどは、相互に情報交換するプログラミングができるように設計されています。
- アプリケーションが期待したとおりに動作しないときは、プログラムを調べ、その原因と思われるステートメントを探します。次に、疑わしいステートメントにブレークポイントやウォッチ ポイントを設定し、アプリケーションを再実行します。ブレークポイントとウォッチ ポイントについては、この章の「問題のあるステートメントでの実行の停止」で説明しています。
- プログラムが停止したら、デバッグ ウィンドウを使って重要な変数，式，プロパティを調べます。また、イミディエイト ウィンドウで、変数やプロパティに別の値を代入してテストすることもできます。

17.5 変数の明示的な宣言

Visual Basic の強力なデバッグ機能の 1 つが、**OPTION EXPLICIT** ステートメントです。このステートメントを使うと、明確に宣言しない変数は認められなくなります。すべての変数を参照される前に強制的に定義させることができます。

プログラムのモジュール レベルに **OPTION EXPLICIT** ステートメントが記述されている場合、そのモジュール内で宣言されていない変数を参照しようとすると、“変数が宣言されていません”というエラー メッセージが表示され、未宣言の変数が強調表示されます。

OPTION EXPLICIT ステートメントを使うことによって、Basic 特有の、暗黙の変数宣言や変数のデフォルトのデータ型の使用のために起こるいくつかの問題を解決できます。

暗黙の変数宣言

Basic では、あらかじめ変数を宣言しておかなくても、参照するだけでその変数を生成することができます。これは便利な反面、単に変数名をスペル ミスしても、新しい変数が作成されてしまうことを意味します。次の例を見てください。

```
MaxOccurences = 100
MSGBOX STR$(MaxOccurences)        '正常 - 値 100 が表示されます。
MSGBOX STR$(MaxOccurrences)       'エラー - 値 0 が表示されます。
```

変数名のスペルをまちがえると論理エラーになります。ただし、プログラムは実行できることに注意してください。このようなエラーは、プログラムのどの部分にどんなタイミングで影響を及ぼすのか予測できないため、発見が最も困難です。プログラムが大きく複雑になるに従って、このような論理エラーの修正には膨大な時間と労力が必要になります。

暗黙の変数宣言は、短いプログラムでは便利な機能で、十分に注意して使用すれば、思考を中断せずにプログラミングできる手段として役立ちます。しかし一般的には、変数はすべて明示的に宣言すべきです。

変数のデフォルトのデータ型

従来の Basic のもう 1 つの便利な機能に、変数のデフォルトのデータ型があります。宣言しないで変数を参照すると、その変数はデフォルトのデータ型で生成されます。デフォルトのデータ型は、**DEF***type* ステートメントの設定によって決まります。

型宣言文字 (%, &, !, #, @, $) を変数名に付ければ、特定のデータ型で変数を作成できます。同じ名前でも、型宣言文字が付けられているかどうか、またどの型宣言文字が付けられているかどうかで、別の変数として扱われることに注意してください。型宣言文字が付けられていないときは、その変数はデフォルトのデータ型になります。したがって、宣言されていない変数を型宣言文字を付けずに参照すると、デフォルトのデータ型の新しい変数が生成されることになります。次の例では、エラーが発生します。

```
Msg$ = "Hello, world"
MSGBOX Msg$                 '正常: "Hello, world" が表示されます。
MSGBOX Msg                  'エラー: 実行時またはコンパイル時に、
                            '"型が一致しません" エラーが発生します。
```

文字列型と整数型のようなデータ型の違いであれば構文エラーとなるため、エラーの発見は比較的簡単です。しかし、どちらも同じデータ型のときは(例えば両方とも整数のときなど)、型宣言文字を付けないと発見が難しい論理エラーとなります。

```
MyNumber% = 99
MSGBOX STR$(MyNumber%)          '正常:99 が表示されます。
MSGBOX STR$(MyNumber)           '論理エラー:0 が表示されます。
```

変数の宣言の詳細については、第16章「プロシージャによる構造化プログラミング」を参照してください。

OPTION EXPLICIT ステートメントを使ったデバッグ

OPTION EXPLICIT ステートメントは、デバッグ ウィンドウやイミディエイト ウィンドウ内での変数の参照にも活用できます。例えば、**OPTION EXPLICIT** ステートメントがモジュール中に記述されていないとき、MyNum の値が 0 以外になったときにプログラムの実行を中断したい場合に、まちがって式 MyNumber <> 0 でウォッチ ポイントを設定してしまったとします。このままではタイプ ミスした名前で変数が生成されて、希望する変数が参照されません。しかし、**OPTION EXPLICIT** ステートメントがモジュール中に記述されていれば、ウォッチ ポイント中で未定義の変数を参照するとエラーが発生します。

17.6　問題のあるステートメントでの実行の停止

デバッグ機能を使ってプログラムの動きを調べるとき、必要に応じてプログラムの実行を中断するようにします。必ず発生する明かなバグでは、実行がその時点に達したら [STOP] キーを押してすぐに実行を中断することができます。しかし、バグは必ずしも明らかなものばかりではないため、バグの原因と思われる部分でアプリケーションを停止させたいことがよくあるはずです。

Visual Basic には、プログラムの実行を中断するための機能として、ブレークポイント，ウォッチ ポイント，**STOP** ステートメントの 3 つが用意されています。"ブレークポイント" は、特定のステートメントでプログラムの実行を中断させたいときに使います。"ウォッチ ポイント" は、指定した論理式が、プログラムの実行によって偽から真 (0 以外の値) に変化したときにプログラムの実行を中断する機能です。**STOP** ステートメントは、実行を中断させたい位置に記述します。

Visual Basic は、次のようなときに、そのとき実行中の行でプログラムの実行を中断します。

- ステートメントが、エラー処理ルーチンで処理されないランタイム エラーを起こしたとき。

  [エラー時中断 ON/OFF] コマンドがオンのときも実行は中断されますが、この場合に中断されるのは実際にエラーが発生した行ではなく、エラー処理ルーチンの最初の行です。

- ブレークポイントが設定されている行に実行が到達したとき。
- ある行の実行によって、ウォッチ ポイントとして設定されている式の値が偽から真に変化したとき。
- **STOP** ステートメントが実行されたとき。

[STOP] キーを押してプログラムの実行を中断することもできます。アプリケーションがアイドル状態 (そのとき処理されているイベントがない状態) のときでも、実行を中断することができます。ただし、[STOP] キーを押したときは、実行は特定の行では中断されず、コード ウィンドウには実行開始モジュールのモジュール レベル コードが表示されます。

ランタイム エラーの修正と継続

一部のランタイム エラーは、プログラム入力時の単純なミスによって発生しているはずです。このようなランタイム エラーの修正は簡単です。例えば、あるコマンド ボタン コントロールのプロパティを、Command1.Text として参照したとします。しかし、このコントロールには Text プロパティはないので、“このコントロールには使用できないプロパティです” ランタイム エラーが発生します。一方、別のエラーは、修正するためにはなぜエラーになったのかを理解しなければなりません。図 17.2 は、あるデータ型の変数に異なるデータ型の値が代入されたときに発生したランタイム エラーを示しています。

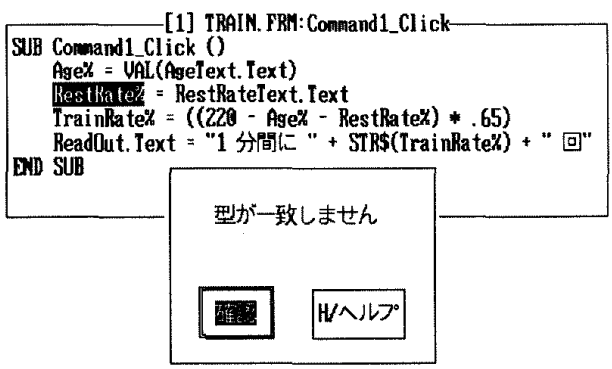


図 17.2 データ型の不一致などのランタイム エラーによる中断

この例では、次のようにエラーが発生しているステートメントを修正し、正しいデータ型 (数値) が変数 RestRate% に代入されるようにします。

```
RestRate% = VAL(RestRateText.Text)
```

プログラムは、中断された場所から継続して実行できます。プログラムの一部を修正した場合でも、多くの場合に実行を続けることができます。実行を継続するには、[実行] メニューから [継続] コマンドを選択するか、[F5] キーを押します。継続実行することで、エラーの原因を取り除くことができたかどうかを確認できます。

変数の宣言を変えた場合など、修正の内容によってはアプリケーションを最初から実行 (リスタート) しなければならないことがあります。このようなときは、継続実行を指定してもアプリケーションのリスタートを求めるメッセージが表示されます。

注　意　リスタートを選択すると、“編集の後、継続できなくなります。よろしいですか?” というメッセージが表示されます。リスタートさせると継続実行はできなくなりますが、それまでの修正内容は保存されます。[ESC] キーを押すか、<取消> ボタンをクリックしてリスタートしなかったときは、それまでの修正内容が破棄されます。

ブレークポイントによる実行の中断

ブレークポイントを設定すると、コードの特定の行を実行する前にプログラムの実行を中断できます。ブレークポイントの設定や解除は、プログラムが実行されていないときならいつでも行うことができます。

▼ ブレークポイントを設定するには

1. コード ウィンドウ中のブレークポイントが設定されていない行にカーソルを移動します。
2. [ﾃﾞﾊﾞｯｸﾞ] メニューから [ブレークポイント ON/OFF] コマンドを選択するか、[F9] キーを押します。

   ブレークポイントが設定された行は、別の色で表示されて、他の行と区別されます。ブレークポイント行の色は、[補助] メニューの [画面設定] コマンドで変更できます。

▼ ブレークポイントを解除するには

1. コード ウィンドウ中のブレークポイントが設定されている ([補助] メニューの [画面設定] コマンドで設定した色で表示されている) 行にカーソルを移動します。
2. [ﾃﾞﾊﾞｯｸﾞ] メニューから [ブレークポイント ON/OFF] コマンドを選択するか、[F9] キーを押します。

   ブレークポイントを解除した行は、設定前の色になります。

[ﾃﾞﾊﾞｯｸﾞ] メニューから [全ブレークポイント解除] コマンドを選択して、すべてのブレークポイントを 1 度に解除することもできます。

図 17.3 では、プロシージャの 3 行目にブレークポイントが設定されており、実行はその行で中断します。この行は強調表示されていて、プログラムを継続して実行するときに次に実行されるステートメントであることを示しています。この行を、“カレント ステートメント” と呼びます。

```
[1] TRAIN.FRM:Command1_Click
SUB Command1_Click ()
    Age% = VAL(AgeText.Text)
    RestRate% = VAL(RestRateText.Text)
    TrainRate% = ((220 - Age% - RestRate%) * .65)
    ReadOut.Text = "1 分間に " + STR$(TrainRate%) + " 回"
END SUB
```

図 17.3 ブレークポイントによって停止したプロシージャ

ウォッチ ポイントによる実行の中断

ウォッチ ポイントは、実行を中断するという点ではブレークポイントに似ています。ブレークポイントは無条件に実行を中断させるために使うのに対し、ウォッチ ポイントは条件付きで (つまり指定した条件が偽から真に変わったときに) 実行を中断させるために使います。

▼ ウォッチ ポイントを設定するには

1. [ﾃﾞﾊﾞｯｸﾞ] メニューから [ウォッチ ポイント] コマンドを選択します。
2. [ウォッチ ポイント] ダイアログ ボックスに式を入力します。
3. [ENTER] キーを押します。

または

1. 既存の式を選択して強調表示させます。
2. [ﾃﾞﾊﾞｯｸﾞ] メニューから [ウォッチ ポイント] コマンドを選択します。

   強調表示した式がデバッグ ウィンドウに表示されます。式は論理的に真または偽と評価されます。

▼ ウォッチ ポイントを解除するには

1. [ﾃﾞﾊﾞｯｸﾞ] メニューから [ウォッチ解除] コマンドを選択します。
2. 解除する式にカーソルを移動します。
3. [ENTER] キーを押します。

   指定した式がデバッグ ウィンドウから消去されます。

STOP ステートメントによるアプリケーションの中断

ブレークポイントを設定する代わりに、プログラム中に **STOP** ステートメントを記述することもできます。実行が **STOP** ステートメントに到達するとプログラムは中断されます。

STOP ステートメントとブレークポイントは、自動的に解除されるかどうかという点で大きく異なります。編集中のプロジェクトを終了するか、Visual Basic を終了してロードし直すと、ブレークポイントはすべて解除されます。一方 **STOP** ステートメントは、ステートメント自身をプログラムから削除しない限り自動的に解除されることはありません。

プログラミング環境でプログラムを実行しているとき、**STOP** ステートメントは、プログラムの実行を一時的に中断させるだけです。[実行] メニューから [継続] コマンドを選択すれば、いつでもプログラムの実行を再開できます。プログラムのデザイン時には、**STOP** ステートメントを使っていろいろな場所でプログラムの実行状態を調べると便利です。

実行可能 (.EXE) ファイルでは、**STOP** ステートメントは **END** ステートメントと同じように動作し、プログラムをただちに終了させます。したがって、**STOP** ステートメントは、実行可能ファイルを作成する前にコードから削除しておきます。

実行を中断したあとの操作

ブレークポイント，ウォッチ ポイント，**STOP** ステートメントによってプログラムの実行は中断されますが、プログラムはアイドル状態にあり、終了しているわけではありません。このとき、オブジェクトはイベントを認識しませんが、中断された時点での状態を調べることができます。デバッグ ウィンドウを使ってデータを調べたり、イミディエイト ウィンドウでデータを変更したり、さらにコード自身を修正することもできます。[ヒストリ ON/OFF] コマンド をオンにしていれば、これまでに実行されたステートメントを 1 つずつさかのぼって実行することも可能です。

プログラムが中断されると、プログラムを中断した行が表示されているコード ウィンドウがフォアグラウンドに表示されると同時に、次の 3 つの情報も表示されます。

- **現在実行しているモジュールの名前**

  コード ウィンドウのタイトル バーに表示されます。

- **現在実行しているプロシージャの名前**

  タイトル バーに、フォーム名かモジュール名に続いて表示されます。

- **次に実行するステートメント**

  カレント ステートメントの色で識別できます。

探している問題がすでに発生しているときは、それまでに実行された行で問題が発生していることになります。まだ発生していないときは、これから実行される行で問題が発生すると判断できます。

実行がブレークポイントに到達すると、アプリケーションはその行が実行される前に中断されます。したがって、ブレークポイントが設定されている行に問題があるときは、少なくともあと 1 ステートメント実行しないと問題を確認できません。ブレークポイントに続くステートメントを実行するには、シングル ステップ実行 ([F8] キー) やプロシージャ ステップ実行([F10] キー) を使います。

ウォッチ ポイントの式の評価が偽から真に変わったためにプログラムが中断されたときは、式の評価を変えたステートメントの直後のステートメントで実行が中断されます。

ヒント 問題と思われる行が見つからなくても、あるステートメントで変数に不正な値が代入されているために問題が発生している場合があります。このようなときは、プログラムが中断されているときに変数とプロパティの値を調べれば、問題の原因を突き止めることができます。

17.7 ウォッチ式の追加

デバッグ ウィンドウに変数や式を表示して、それらの値を監視することができます。変数や式の値は、変数名や式の横に表示されます。ただし、変数や式の値が表示されるのは、変数や式がデバッグ ウィンドウに追加されたとき、画面に表示されていたモジュール レベルのコードやプロシージャが実行されている場合だけです。その他の場合では、デバッグ ウィンドウには値の代わりに <スコープ外> と表示されます。プログラムのいろいろな部分にある非ローカル変数を監視するには、各部分が画面に表示されているときに [ウォッチ追加] コマンドを使います。

デバッグ ウィンドウには、最大で 16 個までのウォッチ ポイントとウォッチ式が入力できます。

▼ **デバッグ ウィンドウで変数や式を監視するには**

1. [ﾃﾞﾊﾞｯｸﾞ] メニューから [ウォッチ追加] コマンドを選択します。
2. ダイアログ ボックスが表示されるので、変数名か式を入力します。

または

1. 変数名か式を選択して強調表示させます。
2. [ﾃﾞﾊﾞｯｸﾞ] メニューから [ウォッチ追加] コマンドを選択します。

変数名や式と、それらの現在の値がデバッグ ウィンドウに表示されます。プログラムの他の部分に実行が移ったときは、その変数名や式の後に <スコープ外> というメッセージが表示されます。

[インスタント ウォッチ] コマンドを実行すると、変数の値や強調表示されていた式を含むダイアログ ボックスが現在のカーソル位置に表示され、その変数や式をデバッグ ウィンドウに表示し続けるかどうかを選択することができます。

モジュール レベルで **SHARED** と宣言されているときなど、変数がスコープ内にあるときでも、その値が表示されるのは、そのモジュール レベルのコードが実行されている場合、または、その変数がデバッグ ウィンドウに追加されたとき画面に表示されていたプロシージャが実行されている場合だけです。

注　意　デバッグ ウィンドウはヘルプ ウィンドウに似ています。すべてのテキストや選択したテキストをデバッグ ウィンドウから印刷できます。また、選択したテキストをクリップボードにコピーすることもできます。ただし、デバッグ ウィンドウに表示されているテキストを編集することはできません。

デバッグ ウィンドウから項目を削除するには、[ﾃﾞﾊﾞｯｸﾞ] メニューの [ウォッチ解除] コマンドか [全ウォッチ解除] コマンドを使用してください。

プロパティの値の監視

デバッグ ウィンドウには、通常の式，プロパティを含んだ式，ウォッチ ポイントなどの有効な式を表示することができます。スコープは現在実行中のフォームやモジュールによって決定されます。あるフォームに対して記述されたコードで実行が停止したときは、そのフォームがカレント フォームになります。そのフォームのコントロールだけでなく、プロパティも表示できます。

カレント フォームのプロパティやコントロールを参照するときは、フォーム名を省略できます。

```
BackColor
Text1.Height
```

Text1 が現在実行中のフォームに配置されたコントロールだとします。最初のステートメントをデバッグ ウィンドウに追加すると、そのプロパティ名とカレント フォームの背景色を示す数値が表示されます。2 番目のステートメントを追加すると、コントロール名，プロパティ名，Text1 の高さが表示されます。

実行がコード モジュールで中断されたときは、次のようにフォーム名を指定しなければなりません。

```
Form1.BackColor
Form1.Text1.Height
```

17.8　アプリケーション中の指定した部分の実行

エラーが発生するステートメントが正確に分かっているときは、ブレークポイントを使ってその位置で実行を中断させることができます。実行が中断したら、状況を確認して必要な修正を行い、プログラムの継続実行またはリスタートを行うことができます。

しかし、通常は問題のありそうな部分を推測することしかできないでしょう。このようなときは、エラーの発生しそうな部分よりも前にブレークポイントを設定しなければなりません。設定したブレークポイントで実行が中断したら、シングル ステップ実行 ([F8]) やプロシージャ ステップ実行 ([F10]) で 1 行ずつ実行しながら、各ステートメ

ントの実行結果を調べることができます。また、ステートメントの実行をスキップしたり、同じステートメントを繰り返し実行させるために、特定の行から実行を開始してプログラムの制御の流れを変更することもできます。

ステートメントをステップ実行して結果を調べることを、トレースと呼ぶことがあります。[トレース ON/OFF] コマンドをオンにすると、ステートメントを 1 ステップずつ実行できます。

ステートメントのシングル ステップ実行

“シングル ステップ実行” は、ステートメントを 1 度に 1 つずつ実行します。各ステートメントをステップ実行した後で、アプリケーションのフォームを調べたり、デバッグ ウィンドウで変数の値を確認したりできます。

▼ **コードを 1 度に 1 ステートメントずつステップ実行するには**

- [F8] キーを押します。

シングル ステップ実行では、1 ステップごとに実行と中断を繰り返しながら、カレント ステートメントが実行されます。実行が中断されると、カーソルは次のステートメントに移動します。カレント ステートメントがプロシージャの呼び出しを行うときにステップ実行を行うと、実行はプロシージャ内に移ります。このとき、次に実行されるステートメント (新しいカレント ステートメント) は、そのプロシージャの 1 行目になります。シングル ステップ実行は [F8] キーでのみ実行でき、メニューからは実行できません。

注 意 ステートメントは、コロン (:) で区切れば 1 行に複数記述できます。シングル ステップ実行では、その行の次に実行されるステートメントだけが強調表示されます。Microsoft CodeView や、その他の多くのデバッグ ツールと異なり、Visual Basic では、同じ行に複数のステートメントが記述されている場合でも、それぞれのステートメントをステップ実行できます。ただし、ブレークポイントは常に行の先頭のステートメントに設定されます。

プロシージャ ステップ実行

“プロシージャ ステップ実行” は、カレント ステートメントがプロシージャを呼び出しているときに使います。カレント ステートメントがプロシージャの呼び出しを行っていなければ、プロシージャ ステップ実行はシングル ステップ実行と同じ動作になります。

▼ **プロシージャ ステップ実行するには**

- [F10] キーを押します。

プロシージャ ステップ実行では、呼び出されたプロシージャを 1 つのステートメントのようにまとめて実行し、そのあとの実行はカレント プロシージャの次のステートメ

ントに移ります。プロシージャ ステップ実行は [F10] キーでのみ実行でき、メニューからは実行できません。

例えば、次のように SetAlarmTime プロシージャを呼び出すステートメントがあるとします。

```
SetAlarmTime 11, 30, 0
```

[F8] キーを押してシングル ステップ実行すると、コード ウィンドウに SetAlarmTime プロシージャが表示され、制御をそのプロシージャの最初のステートメントに移します。SetAlarmTime のコードを分析するときはこの方法が最も適切です。

[F10] キーを押してプロシージャ ステップ実行すると、コード ウィンドウには現在実行中のプロシージャが表示され続けます。SetAlarmTime プロシージャのコードがすべて実行されると、実行は SetAlarmTime プロシージャを呼び出した直後のステートメントに移ります。プログラムの同一レベルで実行を続けたいときや、SetAlarmTime プロシージャの動作を分析しなくてもよいときは、この方法が最も適切です。

シングル ステップ実行とプロシージャ ステップ実行を使い分けると、プログラムのいろいろな部分を分析しながら、プログラム全体の流れを追いかけて把握することができます。

次に実行するステートメントの設定

プログラムのある部分が問題の原因と思われるとき、その部分をスキップして別の行から実行したいことがあります。もちろんこれでは問題の解決にはならないので、最後はその部分をデバッグしなければなりません。しかし、行をスキップすることでアプリケーションの他の部分を実行して分析することができるのであれば、これは有効な方法となります。また、あるステートメントに対し、変数やプロパティの値を変更しながら結果の違いを検討したい場合にも、指定する行からプログラムを実行できると便利です。

このようなとき、Visual Basic では、そのとき実行中のプロシージャ内であれば、別の任意の行から実行を開始することができます。

▼ 次に実行するステートメントを設定するには

1. 次に実行したい行にカーソルを移動します。
2. [ﾃﾞﾊﾞｯｸﾞ] メニューから [次のステートメント設定] コマンドを選択します。

   実行を再開したいときは、[F5] キーを押すか [実行] メニューから [継続] コマンドを選択します。ステップ実行したいときは、[F8] キーか [F10] キーを押します。

ヒストリのトレース

ヒストリ (実行履歴) のオン / オフを切り替えるには、[ﾃﾞﾊﾞｯｸﾞ] メニューから [ヒストリ ON/OFF] コマンドを選択します。ヒストリがオンのときは、プログラムの実行を中断して、プログラムがどのように実行されたのかをさかのぼって調べることができます。[SHIFT]+[F8] キーを押すと、実行されたプログラムの 1 つ前のステートメントにカーソルが移動します。ヒストリは、最大 20 ステートメントまでさかのぼることができます。プログラムの動きを追いかけるのが難しい場合に利用すると便利です。

例えば、フォームの背景色を白に変えたいところで、どうしても紫に変わってしまう場合を考えてみます。このとき、原因が次のステートメントにあるということまではわかりました。

```
BackColor = MyFormColor
```

しかし、これだけでは十分ではなく、このステートメントが実行される前の MyFormColor の値を調べなければ原因ははっきりしません。この値は、[ﾃﾞﾊﾞｯｸﾞ] メニューの [インスタント ウォッチ] コマンドを使えば表示できます。次に、どこで MyFormColor に不正な値が代入されたのかを調べます。プログラムを実行したときに [ヒストリ ON/OFF] コマンドをオンにしてあれば、プログラムを最初からもう 1 度実行しなくても、エラーが発生していそうな部分まで実行をさかのぼることができます。

デバッグ ウィンドウに表示される変数は、プログラムが中断されたときに持っていた値を保持しています。つまり、[ヒストリ ON/OFF] コマンドはプログラムがどのように実行されたのかを表示するだけで、データが変更されることも、次に実行されるステートメントが設定されることもありません。

17.9 イミディエイト ウィンドウの使い方

[ｳｨﾝﾄﾞｳ] メニューから [イミディエイト] コマンドを選ぶと、イミディエイト ウィンドウが画面に表示されます。イミディエイト ウィンドウでは、コードを 1 行だけ入力して、その場で実行できます。1 行に、コロン (:) で区切って複数のステートメントを指定してもかまいません。このウィンドウは、変数の値を変更したり、プログラムの動きをテストするためにプロシージャを呼び出すときなどに利用できます。

▼ イミディエイト ウィンドウでコードを実行するには

1. イミディエイト ウィンドウに、実行したいコードを 1 行入力します。
2. [ENTER] キーを押します。

プログラムを実行した結果、出力があるときは出力画面に表示され、“何かキーを押してください” というメッセージが表示されます。出力画面の内容を見るときは、[F4] キーを押してください。

イミディエイト ウィンドウで、同じ行を複数回実行したいときは、実行したい行にカーソルを移動して [ENTER] キーを押してください。

イミディエイト ウィンドウのスコープ

イミディエイト ウィンドウでは、カレント ステートメントを含むプロシージャやモジュール レベル コードで有効な変数であれば、その変数を操作することができます。操作可能な変数は、第16章「プロシージャによる構造化プログラミング」で示したスコープの規則に従い、そのとき実行されていたプロシージャやフォームによって決まります。

現在中断されているプロシージャのローカル変数にもアクセスできます。ただし、他のフォームやモジュールのローカルな変数にはアクセスできません。

プロパティを指定するときにフォーム名を明確に指定すれば、ロードされているフォームやコントロールのプロパティは、プログラムのどこからでも参照や設定が行えます。次の例では、イミディエイト ウィンドウから次のステートメントを実行すると、プログラムの実行がどこで中断されていても、新しい文字列が Form3 の Text2.Text に表示されます。

```
Form3.Text2.Text = "Help! このテキスト ボックスにはバグがある"
```

注　意　グローバル変数の値は、イミディエイト ウィンドウからいつでもアクセスできます。しかし、イミディエイト ウィンドウから呼び出すことができるイベント プロシージャは、そのときアクティブなフォームのイベント プロシージャだけです。

イミディエイト ウィンドウに入力できるステートメント

イミディエイト ウィンドウには、実行可能ステートメントだけを入力できます。宣言，メタコマンド，その他の非実行可能ステートメントを入力すると、“イミディエイト ウィンドウでは使えません”というエラー メッセージが表示されます。イミディエイト ウィンドウに入力できないキーワードは、次のとおりです。

COMMON	**DIM**	**OPTION EXPLICIT**
CONST	**$DYNAMIC**	**SHARED**
DATA	**$FORM**	**$STATIC**
DECLARE	**$INCLUDE**	**STATIC**
DEF*type*	**OPTION BASE**	**TYPE...ENDTYPE**

注　意　イミディエイト ウィンドウに入力できるのは実行可能ステートメントだけですが、制御構造は、1行で完全に表現できる場合に限り入力できます。したがって、次の **FOR** ループはイミディエイト ウィンドウに入力できます。

```
FOR I = 1 TO 20 : PRINT 2 * I : NEXT
```

17.10 イミディエイト ウィンドウの特性

イミディエイト ウィンドウは、プログラムの実行中でなければいつでも利用することができます。

デバッグ ウィンドウで変数やプロパティを監視しながら、そのコードの一部をイミディエイト ウィンドウで実行すれば、すぐに問題の原因を突き止められる場合があります。イミディエイト ウィンドウでは、値の代入，式の評価，プロシージャの呼び出しなども行えるため、プログラムの中の値を変更しながらその結果を調べたり、あるデータの値が及ぼす影響をテストできます。

変数やプロパティへの値の代入

プログラムが中断されているとき、イミディエイト ウィンドウで代入文を使って変数やプロパティに値を設定できます。

```
BackColor = 255
VScroll1.Value = 100
MaxRows = 50
```

上の例の、最初の 2 つのステートメントでは、現在実行中のフォームのプロパティを変更しています。3 番目のステートメントでは、変数に値を代入しています。中断しているモジュールと別のフォーム モジュールに含まれるフォーム，コントロールやそれらのプロパティを参照したいときは、そのフォームのフォーム名を指定しなければなりません。ローカル変数への代入は、その変数が定義されているプロシージャ内で実行が中断されたときだけ行えます。

プロパティや変数の値を設定したら、実行を継続してその結果を確認することができます。

イミディエイト ウィンドウからのプロシージャの呼び出し

SUB プロシージャや **FUNCTION** プロシージャをイミディエイト ウィンドウから呼び出すことができます。これにより、プロシージャに任意の引数を与え、その結果をテストできます。プロシージャを呼び出すステートメントをイミディエイト ウィンドウに入力して、[ENTER] キーを押してください。プログラムの実行が中断されていてもプロシージャが呼び出され、実行されます。

例えば、次の内容のプロシージャが定義されている場合、イミディエイト ウィンドウから実行することができます。

```
X = Quadratic (2, 8, 8)
DisplayGraph 50, Arr1
```

変数の場合と同じように、プロシージャ呼び出しにもスコープが適用されます。現在実行中のフォーム モジュールに含まれるプロシージャだけでなく、すべてのコード モジュール中のプロシージャも呼び出すことができます。

イミディエイト ウィンドウから適切なイベント プロシージャを呼び出せば、そのイベントをシミュレートすることもできます。ただし、これはカレント フォームだけに利用できます。例えば、次の行は Click イベント プロシージャを直接呼び出して、Click イベントをシミュレートします。

```
Command1_Click
```

イミディエイト ウィンドウからの表示

プログラムが中断されている状態で **PRINT** ステートメントを使うと、変数とプロパティの値をデバッグ ウィンドウではなく出力画面に表示することができます。イミディエイト ウィンドウをアクティブにし、次のように **PRINT** ステートメントで表示したい値を指定します。結果は出力画面に表示され、“何かキーを押してください” というメッセージが画面の一番下の行に表示されます。出力画面の内容は、[F4] キーを押すと見ることができます。

```
PRINT CurrentTotal
```

フォームが表示されている状態で実行を中断したときは、**PRINT** キーワードはメソッドと解釈されて、結果はフォームに表示されます。アプリケーションの実行が終了した状態では、実行がフォーム モジュールで終了したかコード モジュールで終了したかによって、**PRINT** キーワードがメソッドと解釈されるかステートメントと解釈されるかが決まります。フォーム モジュールで終了したときに、値を表示して確認したい場合は、デバッグ ウィンドウを使ってください。

PRINT キーワードの簡略形として、疑問符 (?) が使えます。疑問符は、常に **PRINT** キーワードの代わりに使うことができます。例えば、次の 2 つのステートメントはどちらも同じ意味です。

```
? MyVar
PRINT MyVar
```

イミディエイト ウィンドウで使えるキーとその用途

イミディエイト ウィンドウでは、次のキー操作を利用できます。

- 方向キー

  イミディエイト ウィンドウ内でカーソルを移動するときには、方向キーを使います。カーソルの移動にはマウスも利用できます。

- [ENTER] キー

  1 度イミディエイト ウィンドウに入力したステートメントは、カーソルをそのステートメントに移動して、その行のどこかで [ENTER] キーを押せば繰り返し実行できます。[ENTER] キーを押す前であれば、ステートメントを修正できます。

  実行したいステートメントにカーソルを移動するまでは、[ENTER] キーを押さないでください。

- [PAGE UP] キーと [PAGE DOWN] キー

  1 度に 1 ページ分移動することができます。

■ [HOME] キーと [END] キー

[HOME] キーを押すとカーソルのある行の先頭に、また、[END] キーを押すと行の終わりにカーソルが移動します。[CTRL]+[HOME] キーを押すとイミディエイト ウィンドウの先頭のステートメントに、また、[CTRL]+[END] キーを押すとイミディエイト ウィンドウの最後のステートメントに移動します。

17.11 コマンドライン引数のテスト

実行可能 (.EXE) ファイルでは、起動時にコマンドライン引数を指定できます。プログラミング環境からアプリケーションを実行する場合、コマンドライン引数を直接指定することはできませんが、[実行] メニューの[COMMAND$ 編集] コマンドを使うと、アプリケーションの開発中にコマンドライン引数の引き渡しをシミュレートできます。

オプションで引数を 1 つ指定できるアラーム クロック アプリケーションを作成する場合を考えてみます。アラーム時刻は、起動時にユーザーが直接希望する時刻を入力して指定できるようにします。ユーザーは、次のような文字列を入力してアラーム クロックを起動できます。

```
alarm 11:00:00
```

COMMAND$ 関数は、アプリケーション名の後ろに入力したすべての引数を戻り値として返します。アラーム クロック アプリケーションは引数を 1 つだけとるので、次のように、時刻の値を格納する文字列に直接この引数を代入できます。

```
AlarmTime = Command$
```

COMMAND$ 関数が null 文字列 ("") を返したときは、コマンドライン引数は指定されていないことを表します。このときは、プロンプトを表示してユーザーに入力を求めるようにするか、デフォルトの動作を選択するようにアプリケーションを設計します。

COMMAND$ 関数を使用するプログラムは、プログラミング環境上でコマンドライン引数をシミュレートすることで、動作をテストできます。アプリケーションは、ユーザーがコマンドラインに引数を入力した場合と同じように、このシミュレートされたコマンドライン入力を処理します。

▼ サンプルのコマンドライン引数を設定するには

1. [実行] メニューから [COMMAND$ 編集] コマンドを選択します。
2. テストしたい引数を入力します (アプリケーション名は入力できません)。
3. <了解> ボタンをクリックするか、[ENTER] キーを押します。

17.12　イベントのデバッグに関する注意点

フォームを使ったアプリケーションの開発中は、ユーザーの操作によって対応するイベントが起動されるイベント ドリブン モデルでプログラミングしているということを、いつも考えていてください。プログラムを中断すると、アプリケーションが想定しているイベントの状態が変更される場合があります。

トップ レベルのメニュー クリックに対するプロシージャのデバッグ

トップ レベルのメニューやサブメニューを持つメニュー項目の Click イベント プロシージャにコードを記述する場合、そこで行える操作には制限があります。これは、同様のイベント プロシージャから呼び出されるプロシージャについても当てはまります。詳しくは、第8章「メニューの作成」を参照してください。このようなプロシージャでプログラムの実行を中断したときは、実行を継続することはできず、プログラムをリスタートしなければなりません。

MouseDown イベント中の実行の中断

MouseDown イベント プロシージャ内で実行を中断したときは、マウス ボタンを離したり、マウスで他のいろいろな作業をできるようになります。ただし、実行を継続するときは、アプリケーションはマウス ボタンが押されたままであると想定していることに注意してください。マウス ボタンをもう1度押して離すまで、MouseUp イベントは発生しません。これを避けるには、MouseDown イベント プロシージャにブレークポイントを設定しないようにするか、イミディエイト ウィンドウから MouseUp イベント プロシージャを呼び出します。

KeyDown イベント中の実行の中断

KeyDown イベント中に実行を中断したときは、MouseDown イベントの場合と同じような状況になります。KeyDown イベント プロシージャ中にブレークポイントを設定したときは、KeyUp イベントは得られなくなる場合があります (MouseDown, MouseUp, KeyDown, KeyUp の各イベントについては、第13章「マウス，キーボード，ドラッグ & ドロップ イベント」を参照してください)。これを避けるには、KeyDown イベント プロシージャにブレークポイントを設定しないようにするか、イミディエイト ウィンドウから KeyUp イベント プロシージャを呼び出します。

イベント ドリブン プログラムに特有のバグ

ここまでは、デバッグ中に遭遇する可能性のある問題を説明しました。しかし、イベント ドリブン プログラムの性質をよく理解していないために発生するバグもあります。このようなバグは、いつ、どのようにしてコントロールによりイベントが発生するのかを理解することによって防ぐことができます。

イベントの連鎖反応

イベント プロシージャ内のコードが同じイベントを発生させたときは、そのコードを実行するとイベントが何度も呼び出され、処理が無限に連鎖反応を起こしてしまいます。これは、再帰の深さを制限するコードを持たない再帰プロシージャの動作によく似ています。次のように、テキスト ボックスの Change イベント プロシージャのコードがテキスト ボックスの内容を変更しようとしたときなどによく起こる問題です。

```
SUB Text1_Change()
    Text1.Text = UCASE$(Text1.Text)
END SUB
```

UCASE$ 関数の戻り値が Text1.Text に代入されると、このプロシージャの **END SUB** ステートメントに達する前に Text1 の Change イベントが発生します。この時点で Text1_Change プロシージャはもう 1 度呼び出され、“スタック領域が不足です” エラーが発生するまでこの処理が続きます。

イベントの連鎖反応は、イベント プロシージャが自分自身を呼び出すときだけでなく、いくつかのイベント プロシージャを経由した後で結果的に自分自身が呼び出されるときにも起こります。イベント プロシージャと、そのイベントを発生させるコードが離れているほど、この問題の追跡は困難になります。したがって、どのようなときにイベントの無限連鎖反応が起こる可能性があるのか理解しておかなければなりません。

メニューのクリック イベント プロシージャのコード

メニュー タイトルや、サブメニューを持つメニュー項目では、イベント プロシージャで行える処理には制限があります。第 8 章「メニューの作成」を参照してください。

DOEVENTS 処理

イベント ドリブン アプリケーションのもう 1 つの問題は、**DOEVENTS** 関数が使われたときに起こります。**DOEVENTS** 関数を使うと、大量の計算を行っている間など、集中的に処理を実行しているときに、ユーザーに他の処理を行わせることができます。しかし、ユーザーが次に行った操作が現在実行中の処理と同じ処理を引き起こす場合、その結果は予測できません。通常はイベントの連鎖反応と同じ状態になり、“スタック領域が不足です” エラーが発生します。

先に動作している処理の実行中には、2 回目の処理が実行されないようにすることで、この問題を解決できます。これには、2 回目の処理を実行する前に、1 回目の処理が完了しているかどうかをチェックするコードを記述します。同様に、ある処理がコマンド ボタンを押すかメニュー項目を選択することで開始されたときは、処理が終了するまでそのコントロールを無効にしてください。

コントロールの動作の理解不足

コントロールの動作は、単純なものから複雑なものまでさまざまです。チェック ボックスの動作は単純です。チェック ボックスをクリックすると、チェック ボックスに X マークが表示され、Click イベントが発生します。X マークが付いているときにチェック ボックスをクリックすると、再び Click イベントが発生し、X マークは消えます。

この動作は非常に簡単であり、チェック ボックスでイベントの連鎖反応が起きることはまずありません。

ドライブ リスト ボックス，ディレクトリ リスト ボックス，ファイル リスト ボックスなどの複雑なコントロールは、互いに情報をやりとりするように設計されているため、ある状況では複数のイベントを発生させます。これらのコントロールを使うときには、第10章「ファイル システム コントロールの利用」で説明したように、各コントロール間で同期を取るようにします。コントロール間の情報のやりとりをよく理解するほど、バグの発生を抑えることができます。

注　意　ドライブ リスト ボックス，ディレクトリ リスト ボックス，ファイル リスト ボックスの各コントロールは、ファイル システムに関する情報を調べて表示しますが、カレント ドライブやカレント ディレクトリを実際に変更するわけではありません。例えば、C:¥BASIC ディレクトリから処理を開始し、これら3つのコントロールを使って A:¥PROGRAMS¥BASFILES ディレクトリ中のファイルを表示することができますが、MS-DOS レベルでのカレント ディレクトリは C:¥BASIC のままです。ユーザーに対して表示されている状態は、これら3つのコントロールからの情報を結合すれば得られます。MS-DOS レベルでのカレント ドライブやカレント ディレクトリを実際に変更するには、**CHDRIVE** と **CHDIR** ステートメントを使用してください。

17.13　デバッグにおけるその他の問題点

ここでは、アプリケーションのバグの回避に関して、注意すべきその他の問題点について説明します。

同じ名前を持つ変数のスコープ

一般に、プロシージャ内で宣言された変数 (ローカル変数) は、他のプロシージャ中の同じ名前の変数や、モジュール レベルで **SHARED** キーワード付きで宣言されていない変数とは無関係です。従来の Basic では、モジュール レベルで **SHARED** キーワードを使って宣言された変数名は、プロシージャ内で **STATIC** として宣言されているときのみ、そのプロシージャ内に対してローカルな変数として使うことができました。Visual Basicでは、**DIM** ステートメントを使ってプロシージャ レベルで再び変数を宣言すると、モジュール レベルで **SHARED** キーワード付きで行った宣言より優先されます。

注　意　Visual Basic では、フォーム モジュールのモジュール レベルでは実行可能コードを記述できないので、モジュール レベルで **SHARED** キーワードを指定せずに変数を宣言しても意味がありません。したがって、これによってエラーやバグが発生することはありませんが、プログラムが読みにくくなります。

AS ANYとして宣言されたパラメータ

モジュール内で新しいプロシージャを作成してそのモジュールを保管すると、作成したプロシージャの **DECLARE** ステートメントが、モジュール レベルに自動的に生成されます。例えば、次のように定義したとします。

```
SUB AddItem (Name AS STRING)
```

モジュールを保管すると、次のような **DECLARE** ステートメントが生成されます。

```
DECLARE SUB AddItem (Name AS STRING)
```

ただし、パラメータにユーザー定義型を持つ新しいプロシージャをモジュール内で作成すると、生成される **DECLARE** ステートメントは異なったものになり、そのプロシージャを呼び出したときに問題が発生します。例えば、次のように定義したとします。

```
SUB AddItem (Name AS EMPLOYEE_RECORD)
```

モジュールを保管すると、次のような **DECLARE** ステートメントが生成されます。ユーザー定義型が **ANY** に変わっていることに注意してください。

```
DECLARE SUB AddItem (Name AS ANY)
```

AS ANY は、このプロシージャを呼び出すとそのデータ型が無効になる可能性があることを意味します。**DECLARE** ステートメントを変更し、**AS ANY** を正規のユーザー定義型 (例えば AS EMPLOYEE_RECORD) に書き換えれば、この問題は解決できます。ユーザー定義型をパラメータとしてプロシージャに渡すときは、プロジェクト中のすべてのファイルから文字列 **AS ANY** を検索し、それをすべて変更するようにしてください。インクルード ファイルは、[テキスト] オプションを選択して [ファイル追加] コマンドを実行すれば一時的にテキストとしてロードできます。検索を行うときは、必ず [検索] メニューの [検索] ダイアログ ボックスの [全モジュール] オプションを選択しておきます。

18 Quick ライブラリの作成

作成したプロシージャをソース モジュール (.BAS ファイルや .FRM ファイル) のままにしておくと、いちいちロードしなければならず、メモリを消費すると同時にロードする時間もかかります。さまざまなプログラムから汎用的に使えるモジュールは、Quick ライブラリにしておくと便利です。この章では、Quick ライブラリの概要と Quick ライブラリの作り方、Visual Basic のアプリケーションでの Quick ライブラリの使い方について説明します。

18.1 Quick ライブラリとは

Quick ライブラリは、プログラミング環境 (VBDOS.EXE) でのみ利用できる特別なライブラリです。Visual Basic で作成した汎用的なプロシージャを集めたモジュールから Quick ライブラリを作成し、その Quick ライブラリをプログラミング環境 (VBDOS.EXE) に読み込むと、**SUB** プロシージャは Visual Basic が用意しているステートメントのように、**FUNCTION** プロシージャは関数のように扱うことができます。また、フォーム モジュールを Quick ライブラリに組み込めば、その中のフォームやコントロールを利用できます。Quick ライブラリには、Visual Basic で作成したプロシージャだけでなく、Microsoft C や Microsoft Macro Assembler など、ほかの言語で記述したルーチンを含めることもできます。Quick ライブラリをプログラミング環境に読み込むには、起動時に /L オプションを使って、読み込む Quick ライブラリを指定します。

VBDOS /L *quicklibraryname*[**.QLB**]

Quick ライブラリには次のような使い方があります。

- Visual Basic と他の言語とのインターフェイスとなります。他の言語で作成したオブジェクト モジュールを Quick ライブラリに含めることができます。
- 所有権のあるソフトウェアを保護できます。アップデートするときやユーティリティを提供するとき、Quick ライブラリの形式で配布すれば、ソース コードを保護できます。
- 多くの場合、Quick ライブラリにすることによって、サイズを小さくできます。また、ソース モジュールより高速にロードできます。複数のモジュールを使っていてロードに時間がかかるときは、一部のモジュールを Quick ライブラリにしておけば、ロードの時間を短縮できます。

Quick ライブラリはモジュールの集まりです。つまり、モジュール内の特定のプロシージャだけを Quick ライブラリに含めることはできません。既存のモジュール内の一部のプロシージャだけを使いたいときは、必要なものだけを新しく作成したモジュールにコピーして、そのモジュールを Quick ライブラリに組み込みます。

個々の Quick ライブラリは、それ自身で完結していなければなりません。つまり、Quick ライブラリ中のプロシージャからは、同じ Quick ライブラリ中のプロシージャしか呼び出せません。また、1 つの Quick ライブラリには、同じ名前の複数のプロシージャ名を含めることはできません。

Quick ライブラリとオブジェクト モジュール ライブラリ

Visual Basic で扱うライブラリには、Quick ライブラリとオブジェクト モジュール ライブラリの 2 種類があります。Quick ライブラリの拡張子は .QLB，オブジェクト モジュール ライブラリの拡張子は .LIB です。Quick ライブラリは、プログラミング環境に読み込んで、インタープリタで実行するときに使うのに対し、オブジェクト モジュール ライブラリは、実行可能 (.EXE) ファイルを作成するときにリンクされます。通常、ライブラリを作成するときには、同じモジュールから、この 2 種類のファイルを同時に作成するようにします。このときファイル名は、同じベース名にして、拡張子だけが異なるようにします。プログラミング環境の [実行] メニューの [ライブラリ作成] コマンドを利用すると、この 2 種類のライブラリが自動的に生成されます。

デフォルトの Quick ライブラリ (VBDOS.QLB)

Visual Basic には、VBDOS.QLB というデフォルトの Quick ライブラリが用意されています。また、対応するオブジェクト モジュール ライブラリである VBDOS.LIB も用意されています。Quick ライブラリ名を指定せずに /L オプションだけを指定して Visual Basic を起動すると、この VBDOS.QLB が自動的にロードされます。このファイルには **Absolute, Interrupt, InterruptX** などのルーチンが含まれています。

アプリケーションでこれらのルーチンを利用するときは、プログラミング環境の起動時に、VBDOS.QLB を必ずロードします。ただし、1 度にロードできる Quick ライブラリは 1 つだけなので、別のモジュールも Quick ライブラリに含めたいときは、VBDOS.LIB を使って、これらのルーチンも含む Quick ライブラリを作成してください。

コモン ダイアログの Quick ライブラリ (CMNDLG.QLB)

Visual Basic のサンプルとして用意されているコモン ダイアログ ツールキットにも、Quick ライブラリ (CMNDLG.QLB) が用意されています。プログラミング環境で、作成するアプリケーションからコモン ダイアログ ツールキットを利用したいときには、この Quick ライブラリをロードしてください。

18.2 Quick ライブラリのロードと表示

ここでは、Visual Basic の起動時に Quick ライブラリをロードする方法と、その内容を表示する方法を説明します。また、Quick ライブラリに含まれるプロシージャが浮動小数点演算を行うときの注意点についても説明します。

Quick ライブラリのロード

Quick ライブラリをロードするには、Visual Basic の起動時にコマンド ラインで Quick ライブラリ名を指定します。プログラミング環境に 1 度にロードできる Quick ライブラリは 1 つだけです。構文は次のとおりです。

VBDOS /L *quicklibraryname*[**.QLB**]

/L の後に、ロードする Quick ライブラリの名前 (*quicklibraryname*) を指定します。拡張子 .QLB は省略できます。/L だけを指定してライブラリ名を省略すると、デフォルトの Quick ライブラリである VBDOS.QLB がロードされます。Quick ライブラリがロードされると、その Quick ライブラリに含まれるプロシージャをプログラミング環境で利用できるようになります。

例えば、Visual Basic 起動時に、コモン ダイアログ ツールキットを利用するアプリケーション NEW.BAS を指定するには、コマンドラインに次のように入力します。

```
VBDOS NEW.BAS /L CMNDLG.QLB
```

DECLARE ステートメントで必要なプロシージャを宣言すると、CMNDLG.QLB に含まれるすべてのプロシージャが、アプリケーションから使えるようになります。

Quick ライブラリが探されるディレクトリ

/L オプションで Quick ライブラリが存在するディレクトリを明確に指定すると、そこからロードされます。指定しなければ、次の順番で検索されます。

1. カレント ディレクトリ。
2. [補助] メニューの [パス設定] ダイアログ ボックスで設定されている、ライブラリを含むディレクトリ。
3. 環境変数 LIB で指定されたディレクトリ。

環境変数については、MS-DOS のマニュアルを参照してください。

Quick ライブラリの内容の表示

Quick ライブラリはバイナリ形式のため、テキスト エディタでその内容を表示することはできません。Visual Basic のパッケージに含まれているサンプル アプリケーションの QLBVIEW を使えば、指定したライブラリ中のすべてのプロシージャを表示できます。QLBVIEW は、Visual Basic のセット アップ時にサンプル アプリケーション用として指定したディレクトリにインストールされます。

▼ Quick ライブラリの内容を表示するには

1. MS-DOS のプロンプトで次のように入力して Visual Basic を起動し、QLBVIEW アプリケーションをロードします。

   ```
   VBDOS QLBVIEW
   ```

2. [実行] メニューから [スタート] を選択します (または [SHIFT]+[F5] を押します)。

   Quick ライブラリ ビューアが表示されます。

3. [オープン] ボタンを選択して [オープン] ダイアログ ボックスを表示し、オープンしたい Quick ライブラリを選択します。

   選択した Quick ライブラリに含まれるプロシージャがリスト ボックスに表示されます。

18.3 Quick ライブラリの作成

Quick ライブラリは、[実行] メニューの [ライブラリ作成] ダイアログ ボックスを使って作成します。また、コマンド ラインから BC コンパイラ, LIB ユーティリティ, LINK ユーティリティを直接実行して作成することもできます。

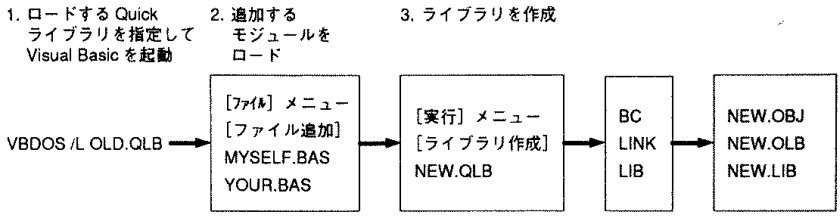
プログラミング環境内から Quick ライブラリを作成すると、Visual Basic の起動時にロードした他の Quick ライブラリの内容も含め、その時点でプログラミング環境にロードされているすべてのモジュールが Quick ライブラリに自動的に組み込まれます。プロジェクトに含まれるモジュールのうち、特定のモジュールだけを Quick ライブラリに組み込みたいときは、プロジェクト全体をロードしてから、[ファイル解放] コマンドで不必要なモジュールを解放してください。現在どのモジュールがロードされているかを調べるには、[表示] メニューから [コード選択] コマンドを選択します。

例えば、既存の Quick ライブラリ OLD.QLB に、コード モジュール MYFILE.BAS および YOUR.BAS に含まれるプロシージャを追加して、新しい Quick ライブラリを作成したいときは、次のような手順になります。

1. /L オプションで Quick ライブラリ OLD.QLB を指定して、Visual Basic を起動します。

2. [ﾌｧｲﾙ] メニューの [ファイル追加] コマンドでモジュールを追加します。または、[ﾌｧｲﾙ] メニューの [プロジェクトオープン] コマンドでプロジェクト全体をロードしてから、[ﾌｧｲﾙ] メニューの [ファイル解放] コマンドで不必要なモジュールを解放します。

3. [実行] メニューの [ライブラリ作成] コマンドを選び、Quick ライブラリとオブジェクト モジュール ライブラリを作成します。



ライブラリの作成を開始すると、BC.EXE, LINK.EXE, LIB.EXE の 3 つのユーティリティが呼び出され、モジュールのコンパイルとリンクを行い、Quick ライブラリ (.QLB) とオブジェクト モジュール ライブラリ (.LIB) を作成します。これらのユーティリティはコマンド ラインから別々に起動することも可能ですが、次のように使い分けます。

- 既存の Quick ライブラリやモジュールから Quick ライブラリを作成するときは、どちらの方法でも行えます。ただし、プログラミング環境から実行すると、適切なオプション設定でユーティリティが自動的に呼び出されるため、作業が簡単になります。
- 他の言語で記述されたオブジェクト ファイル (.OBJ) やオブジェクト モジュール ライブラリから Quick ライブラリを作成するときは、必ずコマンド ラインから LINK などを直接呼び出すようにします。

これ以降では、両方の使い方について説明します。BC, LINK, LIB の 3 つのユーティリティの構文とオプションについては、オンライン ヘルプを参照してください。

Quick ライブラリの作成に必要なファイル

Quick ライブラリを作成するときは、次のファイルがカレント ディレクトリにあるか、MS-DOS の環境変数 (PATH または LIB) で設定されているディレクトリにあることを確認してください。

ファイル	目的
VBDOS.EXE	プログラミング環境。Quick ライブラリの作成処理を管理します。Visual Basic のモジュールと既存の Quick ライブラリだけを使用するときは、プログラミング環境内で必要な処理を簡単に行うことができます。
BC.EXE	Basic コンパイラ。ソース モジュールからオブジェクト ファイルを作成します。
LINK.EXE	リンカ。オブジェクト ファイルをリンクします。
LIB.EXE	ライブラリアン。オブジェクト モジュール ライブラリを管理します。
VBDOSQLB.LIB	Quick ライブラリの作成に必要なルーチンが含まれています。Quick ライブラリに組み込むオブジェクト ファイルとリンクされて、Quick ライブラリを生成します。
VBDCL10.LIB	Visual Basic の標準ライブラリ。

また、/L オプションを指定して Quick ライブラリをロードしているときには、同じ内容のオブジェクト モジュール ライブラリ (.LIB) も必要になります。

Quick ライブラリに組み込むことができるファイル

Quick ライブラリは、次の 4 種類のファイルを基に作成できます。

- コード モジュール (.BAS)
- フォーム モジュール (.FRM)
- オブジェクト モジュール (.OBJ)

  Visual Basic で作成したモジュールをコンパイルしたものだけでなく、C や Macro Assembler など、マイクロソフトの別の言語で記述したものでもかまいません。
- オブジェクト モジュール ライブラリ (.LIB) ファイル

注　意　作成した Quick ライブラリを利用するアプリケーションが、従来の Basic からサポートされているイベントのトラップの機能 (Visual Basic のイベント ドリブン プログラミング モデルとは異なります) を使う可能性があるとき、Quick ライブラリに組み込む最低 1 つのモジュールは、イベントをトラップするステートメントが含まれている必要があります。このステートメントは **TIMER OFF** などの単純なものでかまいません。このようなステートメントがない場合、Quick ライブラリでイベントのトラップが正しく行われません。

既存の Quick ライブラリの使い方

QuickBasic や Microsoft Basic Professional Development System で作成された既存の Quick ライブラリを、Visual Basic でそのまま利用することはできません。その Quick ライブラリを作成するときに使ったソース モジュールを基に、Quick ライブラリを作成し直してください。

拡張子 .QLB

Quick ライブラリには通常、拡張子 .QLB が付けられますが、他の拡張子を付けても、拡張子を付けなくてもかまいません。ただし、/L *libraryname* オプションで拡張子を省略すると、拡張子は .QLB であると見なされます。拡張子のない Quick ライブラリを指定するときは、Quick ライブラリ ファイルのベース名のあとに必ずピリオド (.) を付けます。

プログラミング環境からの Quick ライブラリの作成

プログラミング環境内から Quick ライブラリを作成するときは、まず新しいモジュールから作成するのか、既存の Quick ライブラリ中のプロシージャを含めて作成するのかを決めます。

- 新しいモジュールから Quick ライブラリを作成するには、コマンド ラインから /L オプションを付けずに Visual Basic を起動します。

  VBDOS

- 既存の Quick ライブラリのルーチンを含めるには、コマンド ラインから /L オプションで既存の Quick ライブラリ名を指定して Visual Basic を起動します。

```
VBDOS /L MYLIB.QLB
```

組み込むモジュールのロード

メモリ上に、Quick ライブラリに組み込みたいモジュールだけがあるようにします。必要なモジュールを 1 つずつロードしてもかまいませんし、プロジェクト全体をロードしてから、不必要なモジュールをメモリから解放してもかまいません。

▼ モジュールをロードするには

- [ﾌｧｲﾙ] メニューから [ファイル追加] コマンドを選択します。

▼ モジュールを解放するには

- [ﾌｧｲﾙ] メニューから [ファイル解放] コマンドを選択します。

[ライブラリ作成] ダイアログ ボックスの使い方

組み込みたいモジュールをロードしたら、Quick ライブラリを作成します。

▼ Quick ライブラリを作成するには

1. [実行] メニューから [ライブラリ作成] コマンドを選択します。
2. [ライブラリ名] テキスト ボックスにライブラリ名を入力します。

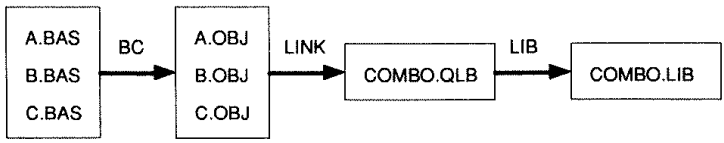
   ベース名 (ファイル名の拡張子以外の部分) だけを入力したときは、ライブラリの作成時に拡張子 .QLB が自動的に付けられます (MYLIB.QLB)。サブディレクトリを指定すると、そのサブディレクトリにライブラリが作成されます。
3. ランタイム エラー チェックのためのデバッグ コードを生成したいときは、/D オプションを選択します。

   /D オプションを選択すると、Quick ライブラリ内のプロシージャの実行中に [STOP] キーを押したときにプログラミング環境に戻ることができます。
4. <ライブラリ作成> ボタンを選択して Quick ライブラリの作成を開始します。Quick ライブラリの作成後に MS-DOS に戻りたいときは、<ライブラリ作成後終了> ボタンを選択します。

作成されるその他のファイル

Visual Basic 環境内からライブラリを作成すると、拡張子 .OBJ と .LIB を持つ 2 つのファイルが作成されます。オブジェクト モジュール ライブラリは実行可能 (.EXE) ファイルを作成するときに利用されるもので、Quick ライブラリと同じオブジェクト モジュールを含んでいます。同じモジュールから生成されたこの 2 種類のライブラリを、それぞれ対応するライブラリと呼びます。この 2 つのライブラリは 1 組として管理し、片方に変更を加えたら、もう片方にも同じ変更を加えるようにしてください。ライブラリの作成や管理をプログラミング環境からだけ行っていれば、この 2 つのライブラリは常に対応が取れています。

この他にも、各モジュールをコンパイルした結果であるオブジェクト ファイル (.OBJ) も生成されます。ファイル名は、ロードしたモジュールと同じベース名に、拡張子 .OBJ を付けたものです。このファイルは、ライブラリ作成後に削除してもかまいません。



ソース モジュールと既存の Quick ライブラリから作成

既存の Quick ライブラリにモジュールを追加して、別の Quick ライブラリを作成することもできます。OLD.QLB という名前の Quick ライブラリを作成したとします。この既存のライブラリにコード モジュールNEW.BAS に含まれるプロシージャを追加して、COMBO.QLB という新しい Quick ライブラリを作成したいときは、次の手順で行います。

1. Visual Basic の起動時に Quick ライブラリをロードします。

   ```
   VBDOS /L OLD.QLB
   ```

   ここでは Quick ライブラリをロードしますが、実際にライブラリを作成するときには、対応する (同じベース名の) オブジェクト モジュール ライブラリ (.LIB) が使われます。このファイルは、カレント ディレクトリ，[補助] メニューの [パス設定] コマンドで設定されているディレクトリ，または LIB 環境変数で設定されているディレクトリになければなりません。この例では、OLD.LIB が必要です。

2. [ﾌｧｲﾙ] メニューから [ファイル追加] コマンドを選択して、NEW.BAS を選択します。
3. [実行] メニューから [ライブラリ作成] コマンドを選択します。
4. [ライブラリ名] テキスト ボックスに COMBO と入力します。
5. [ライブラリ作成] ボタンを選択して Quick ライブラリを作成します。

これで、OLD.QLB と NEW.BAS 中のすべてのプロシージャが結合されて、新しい Quick ライブラリ COMBO.QLB が作成されます。

コマンドラインからの Quick ライブラリの作成

Quick ライブラリは、コマンドラインから直接 BC, LINK, LIB の 3 つのユーティリティをそれぞれ起動しても作成できます。例えば、コモン ダイアログを構成する 2 つのファイル CMNDLGF.FRM と CMNDLG.BAS から Quick ライブラリを作成するには、次の手順で行います。

1. フォーム モジュール CMNDLGF.FRM とコード モジュール CMNDLG.BAS を Basic コンパイラ (BC.EXE) でコンパイルします。

   ```
   BC /X CMNDLGF.FRM;
   BC /X CMDFLG.BAS;
   ```

2. 生成されたオブジェクト ファイル CMNDLGF.OBJ と CMNDLG.OBJ を、リンカ (LINK.EXE) に /Q オプションを指定してリンクします。/Q オプションは、Quick ライブラリを作成することを示します。

```
LINK /Q CMNDLGF.OBJ CMNDLG.OBJ, CMNDLG.QLB,, VBDOSQLB.LIB;
```

リンクするライブラリとして、必ず VBDOSQLB.LIB を指定しなければならないことに注意してください。

3. 同じオブジェクト ファイルから、ライブラリアン (LIB.EXE) でオブジェクト モジュール ライブラリ CMNDLG.LIB を作成します。

```
LIB CMNDLG.LIB+CMNDLGF.OBJ+CMNDLG.OBJ;
```

同名のオブジェクト モジュール ライブラリ (この場合は CMNDLG.LIB) がすでに存在すると、LIB は "module redefinition ignored." というエラー メッセージを表示します。同名のファイルがあるときは、ファイル名を変更するか、あらかじめ削除しておいてください。

注 意 Quick ライブラリは far メモリ (コンベンショナル メモリ) に格納されるため、サイズをできる限り小さくするように注意してください。EMS メモリに十分な容量を確保できるときは、ソースモジュールは EMS メモリに格納されるため、Quick ライブラリにせずソース モジュールのままにしておいた方が、より大規模なプログラムを作成できることがあります。

Basic 以外の言語モジュールからの Quick ライブラリの作成

Basic 以外の言語で書かれたルーチンを Quick ライブラリに組み込むには、まずコンパイラまたはアセンブラで、オブジェクト ファイルを作成してください。また、他の言語で記述したプログラムは、ミディアム メモリ モデルでコンパイルされている必要があります。

次の言語で記述したルーチンが利用できます。

- Microsoft C/C++ version 7.0A
- Microsoft Macro Assembler version 6.0

注 意 Basic で作成したアプリケーションに組み込んで使うルーチンを他の言語で記述するには、さまざまな注意が必要です。それぞれの言語製品に付属のマニュアルの、「複数言語によるプログラミング」の項目を参照してください。また、『リファレンス』の **DECLARE** ステートメントや、オンライン ヘルプを参照してください。

Basic 以外の言語で記述したルーチンを Quick ライブラリに組み込む場合、プログラミング環境からは操作できないため、必ずコマンドラインから直接 LINK や LIB を起動して作成しなければなりません。LINK や LIB のコマンドライン オプションについては、各言語製品のマニュアルやオンライン ヘルプを参照してください。

18.4 Quick ライブラリのプログラミング上の注意

Quick ライブラリは、Visual Basic や、他のマイクロソフトの言語で記述することができます。ここでは、Quick ライブラリの開発に役立つ情報について説明します。

B_OnExit ルーチン

Visual Basic は、Basic のシステム レベルの関数 **B_OnExit** をサポートしています。他の言語で書いたルーチンが、プログラムの終了や再実行の前に元の状態に戻す必要のある特殊な動作を行う場合、**B_OnExit** を使ってください。例えば、Visual Basic のプログラミング環境で、Quick ライブラリ中の他言語のルーチンを呼び出すプログラムを実行するとき、必ずしも正常に終了するとは限りません。このような他言語のルーチンが終了時に特別な動作を行う必要があるときは (例えば前にインストールした割り込みベクタを解放するなど)、ルーチン中に **B_OnExit** を組み込んでおくと、終了ルーチンが確実に呼び出されるようになります。

B_OnExit の使用例を示します。この例では、わかりやすくするためにエラー処理コードは省略されています。このような関数は、Microsoft C のミディアム モデルでコンパイルします。

```
<malloc.h>
extern pascal far B_OnExit();        /* ルーチンの宣言 */
int *p_IntArray;
void InitProc()
{

    void TermProc();                 /* TermProc 関数の宣言 */
    /* 20個の int 型配列に対する far スペースの割り当て: */
    p_IntArray = (int *)malloc(20*sizeof(int));
    /* Basic の終了ルーチン (TermProc) のログ */
    B_OnExit(TermProc);
}
                         /* TermProc 関数は、プログラムの再実行前  */
void TermProc()          /* または終了前に呼び出されます*/
{
    free(p_IntArray);    /* InitProc が割り当てた far スペース  */
                         /* の解放*/
}
```

InitProc 関数が Quick ライブラリ内にある場合、プログラムがクラッシュして **B_OnExit** が呼び出されたとき、malloc 関数の呼び出しで確保された領域は確実に解放されます。プログラムは Visual Basic の環境から何回か実行されるので、このルーチンも何回か呼び出されます。しかし、TermProc 関数自身は、プログラムが実行されるごとに 1 度だけ呼び出されます。

次の Basic のプログラムは、InitProc 関数を呼び出す例です。

```
DECLARE SUB InitProc CDECL

    X = SETMEM(-2048)            'C 関数での malloc 関数による
                                 'メモリ割り当て用の領域を確保

    CALL InitProc
END
```

B_OnExit で登録できるルーチンは 32 までです。33 以上のルーチンを登録すると、**B_OnExit** は登録のための空間がないことを表す NULL を返します (**B_OnExit** の戻り値は、Microsoft C のランタイム ライブラリ ルーチン onexit と同じです)。

B_OnExit は、Quick ライブラリに組み込まれた他の言語 (アセンブリ言語を含む) のルーチンでも使用できます。実行可能ファイルを作成するときには、**B_OnExit** を使う必要はありません。

文字列と配列の扱い

Quick ライブラリに組み込む他言語のルーチンを記述するとき、そのルーチンで Basic の文字列を処理する場合は、Basic の far 文字列ルーチンを使用する必要があります。詳細は、オンライン ヘルプまたは『リファレンス』の StringAddress, StringAssign, StringLength, StringRelease の各ルーチンのトピックを参照してください。なお、これらのルーチンを Basic から利用するには、何らかの Quick ライブラリを読み込んでおく必要があります。

Quick ライブラリに、Basic で生成した配列を受け取る他言語のルーチンが含まれている場合は、/Ea (EMS) オプションを指定して Visual Basic を起動することはできません。

コンパクトな実行可能ファイルの作成

Quick ライブラリを作成すると、それに対応するオブジェクト モジュール ライブラリも作成されます。プログラミング環境に Quick ライブラリがロードされている場合、実行可能 (.EXE) ファイルを作成すると、対応するオブジェクト モジュール ライブラリから必要なオブジェクト モジュールが検索されます。

通常、1 つのライブラリは複数のオブジェクト モジュールから構成されています。モジュール中のプロシージャがプログラムから 1 度も参照されていなければ、そのオブジェクト モジュールは実行可能ファイルに組み込まれません。しかし、1 つでもプロシージャが参照されていると、実行可能ファイルにはそのプロシージャを含むモジュール全体が組み込まれてしまいます。例えば、あるモジュール中に含まれる 4 つのプロシージャのうちの 1 つしか使用されない場合でも、そのモジュール全体が実行可能ファイルに組み込まれます。

このような場合、実行可能ファイルのサイズをできるだけ小さくするためには、各モジュールに関連のあるプロシージャだけを含めるようにライブラリを再構成しなければなりません。ライブラリの内容を調べたいときは、この章の「Quick ライブラリの内容の表示」で説明したサンプル アプリケーション QLBVIEW を使ってその内容を表示してください。

付録

A ASCII フォームのフォーマット

Visual Basic では、モジュールを保管するファイルの形式にバイナリ形式とテキスト形式の 2 種類があります。フォーム モジュールは、デフォルトではバイナリ形式で保管されますが、テキスト形式で保管することも可能です。テキスト形式で保管されたフォーム モジュールを、ASCII フォームと呼びます。ASCII フォームとして保管すれば、Visual Basic 以外のエディタやツールでも、フォームに関するデータを扱うことができます。また、この機能を利用することによって、フォームの自動作成ツールなども作成することが可能です。Microsoft CodeView や Source Profiler でフォーム モジュールを扱うには、ASCII フォームとして保存してから /Zi または /Zd オプションを付けてコンパイルしてください。

▼ **フォーム モジュールを ASCII フォームとして別ファイルに保存するには**

- [ﾌｧｲﾙ] メニューから、[新規保管] コマンドを選択し、[テキスト形式] オプションを指定します。

注　意　フォーム デザイナからテキスト形式で保管できるのは、まだコードが記述されていないフォーム モジュールだけです。コードが記述されているフォーム モジュールをテキスト形式で保管したい場合は、プログラミング環境で行ってください。

A.1 記録される情報

テキスト形式で保管されたフォーム モジュール、つまり ASCII フォームには、次のような情報がテキストとして記録されています。

- バージョン番号 (省略可能)
- フォームに関する情報 (コントロールを含むフォームの外観を記述した文字列ブロック)
- フォームに関連する Basic コード (イベント プロシージャやジェネラル プロシージャ，モジュールレベル コード)

フォームに関する情報としては、フォームやコントロールと、それらのプロパティの設定が、テキストで記述されます。フォームやコントロールの親子の関係はネストによって表現されています。フォームに関する情報の構造を、次に示します。

```
Version 1.0
BEGIN Form フォーム名
    .
    . (フォームのプロパティの情報)
    .
    BEGIN コントロールの種類 コントロール名
        .
        . (コントロールのプロパティの情報)
        .
    END
    BEGIN コントロールの種類 コントロール名
        .
        . (コントロールのプロパティの情報)
        .
    END
END
(フォーム モジュールに記述されている Basic コード)
```

バージョン番号

Visual Basic for MS-DOS で作成した ASCII フォームのバージョン番号は 1.00 です。別のエディタやツールを使って Visual Basic for MS-DOS 用の ASCII フォームを作成するには、正しいバージョン番号を付けるようにしてください。バージョン番号が省略されている場合は、1.00 と見なされます。

フォーム情報

フォームに関する情報は、BEGIN で始まり、END で終わります。フォームに対する BEGIN の行では、次の構文に従ってそのフォームの名前を定義します。

BEGIN Form *FormName*

フォームに関する情報は END の行で終わり、次の行から、フォーム モジュール中の Basic コードが続きます。END の行がないと、Basic コードがフォームのコントロールやプロパティと解釈され、エラーが発生します。

BEGIN の行と END の行の間には、まずフォームのプロパティ、その次にフォームに含まれるコントロールのプロパティが続きます。**FOR...NEXT** ステートメントのように、BEGIN から、その BEGIN に対応する END までが 1 つのブロックになり、オブジェクトの親子関係に従ってネストして示されます。

```
Version 1.0
BEGIN Form MyForm1
    AutoRedraw      = 0
    BackColor       = QBColor(7)
    BorderStyle     = 2
    Caption         = "フォーム タイトル"
    ...
    BEGIN Frame MyFrame1
        BackColor       = QBColor(7)
        Caption         = "Frame1"
        DragMode        = 0
        Enabled         = -1
        ...
        BEGIN CommandButton MyButton1
            BackColor       = QBColor(7)
            Cancel          = 0
            Caption         = "開始"
            Default         = 0
        ...
        END
    END
END
```

コントロール ブロック

フォーム情報の中で、コントロールのプロパティを定義する部分がコントロール ブロックです。1 つのコントロールに対するコントロール ブロックは、フォームと同様に BEGIN の行で始まり、END の行で終わります。コントロールに対する BEGIN の行では、次の構文に従ってそのコントロールの種類と名前を定義します。

Begin *ControlType ControlName*

前の例では、フォーム MyForm1 のフォーム情報の中にフレーム MyFrame1 のコントロール ブロックが含まれ、さらにその中にコマンド ボタン MyButton1 のコントロール ブロックが含まれています。

コントロール ブロックの順番

TabIndex プロパティが指定されていない場合は、フォーム情報内でのコントロール ブロックの順番によって、TabIndex の値が決ります。コントロール ブロックの順番は、コントロールが描画される順番も決定します。コントロールが重なって表示される場合、最後に描画されたコントロールが一番上（手前）に表示されます。

コントロール ブロックのネスト

コントロール ブロックのネストは、コントロールの親子関係を示します。フレーム，ピクチャー ボックス，メニューのコントロール ブロックは、その中に別のコントロールのコントロール ブロックを含むことができます。

例えば、フレームがコンテナとして機能し、その中にオプション ボタンが配置されていれば、これらのコントロール ブロックをネストして示します。コンテナ側のコントロールのプロパティの設定は、子コントロールのプロパティの設定より前に記述します。これは、子コントロールをロードする前に、ロードに必要なすべての情報が得ら

れていなければならないためです。子コントロールより後に記述された親コントロール (コンテナ) のプロパティの設定は無視されます。

メニュー コントロール

すべてのメニュー コントロールは、フォーム情報中でひとまとめにして記述しなければなりません。Visual Basic が ASCII フォームをロードするとき、メニュー コントロールに関する記述を見つけると、すべてのメニュー コントロールがその後に現れるものと仮定されます。いったんメニュー コントロール以外のコントロールが現れると、そのフォームにはそれ以上のメニューコントロールはないと見なされ、それ以降に現れたものについては無視されます。

プロパティ

Visual Basic で作成してテキスト形式で保管した ASCII フォームでは、プロパティはアルファベット順に並べられますが、別のエディタやツールで ASCII フォームを作成する場合は、必ずしもアルファベット順に並んでいる必要はありません。また、そのオブジェクトが持っているすべてのプロパティの情報を記述する必要もありません。記述しなかったプロパティや、プロパティ名だけで値を指定しなかったものに対しては、ロード時にデフォルトの値が設定されます。Visual Basic で作成して保管した ASCII フォームには、すべてのプロパティとその設定値が含まれます。

構文

ASCII フォームでは、プロパティの情報は次の構文に従います。

Property = *Value*

文字列はダブル クォーテーションで囲んで指定します。True または False の値を取るプロパティでは、0 以外の値は True (真) を表し、0 は False (偽) を表します。いくつかの選択肢から 1 つを選んで設定するプロパティには、名前ではなく数値を記述します。例えば BorderStyle プロパティは、ASCII フォームでは次のように記述します。

```
BorderStyle = 2
```

コメント

フォームやコントロール、およびそれらのプロパティに関する記述には、コメントを付けることができます。Basic コードと同じように、シングル クォーテーション(') がコメントを示します。

注 意 Visual Basic でフォーム モジュールをテキスト形式で保管した場合は、フォームやコントロールとそのプロパティに対してはコメントは付きません。Basic コード中のコメントはそのまま残されます。

各プロパティに関する注意

次の各プロパティについては、特別な処理が必要です。

ショートカット キー

メニュー コントロールのショートカット キーを表現するとき、+ が [SHIFT] キー、^ が [CTRL] キー、{F*n*} がファンクション キー (*n* はキーの番号) を表します。アルファベットのキーは、その文字を表します。

```
Shortcut = "^{F4}"
```

Top, Left, Width, Height

画面上の位置を示す文字座標は、CHAR(*x*) のように示します。*x* は桁数または行数です。

```
Top = CHAR(16)
```

CHAR が省略されると、*x* は twip (1/20 ポイント) を表すものと解釈されます。フォームがロードされると、twip は文字単位に変換されます。Visual Basic for MS-DOS のサポートする位置の単位は文字座標のみで、端数は切り捨てられます。

ForeColor, BackColor

基本色は、QBCOLOR(*x*) を使って指定します。*x* は、色番号を示す 0 ～15 の数値です。

```
BackColor = QBCOLOR(3)
```

QBCOLOR が省略されると、*x* は RGB 値と解釈されます。フォームがロードされると、RGB 値は最も近い色の色番号に変換されます。

FormType

FormType プロパティは、そのフォームの BEGIN の行に記述します。MDI アプリケーションのコンテナ フォームでは MDIForm と指定し、それ以外のフォームでは Form と指定します。

```
BEGIN MDIForm Form3
```

Caption

Caption プロパティをメニュー中に置くことで、セパレータ バーを定義することができます。セパレータ バーを示すメニュー コントロールには、Caption プロパティに - (マイナス) を指定します。

```
Caption = "-"
```

Basic コード

ASCII フォーム中の Basic コードは、フォーム情報の中の最後の END の行の直後から始まります。

A.2　ASCII フォームのサンプル

このマニュアルの第 2 章で作成した“Hello, world”サンプル アプリケーションをテキスト形式で保管した ASCII フォームを次に示します。

```
Version 1.00
BEGIN Form Hello
    AutoRedraw    = 0
    BackColor     = QBColor(7)
    BorderStyle   = 2
    Caption       = "Hello, world"
    ControlBox    = -1
    Enabled       = -1
    ForeColor     = QBColor(0)
    Height        = Char(17)
    Left          = Char(15)
    MaxButton     = -1
    MinButton     = -1
    MousePointer  = 0
    Tag           = ""
    Top           = Char(3)
    Visible       = -1
    Width         = Char(63)
    WindowState   = 0
    BEGIN CommandButton Command1
        BackColor     = QBColor(7)
        Cancel        = 0
        Caption       = "OK"
        Default       = 0
```

```
    DragMode      = 0
    Enabled       = -1
    Height        = Char(3)
    Left          = Char(16)
    MousePointer  = 0
    TabIndex      = 0
    TabStop       = -1
    Tag           = ""
    Top           = Char(9)
    Visible       = -1
    Width         = Char(13)
END
BEGIN CommandButton Command2
    BackColor     = QBColor(7)
    Cancel        = 0
    Caption       = "終了"
    Default       = 0
    DragMode      = 0
    Enabled       = -1
    Height        = Char(3)
    Left          = Char(38)
    MousePointer  = 0
    TabIndex      = 1
    TabStop       = -1
    Tag           = ""
    Top           = Char(9)
    Visible       = -1
    Width         = Char(13)
END
BEGIN TextBox Text1
    BackColor     = QBColor(7)
    BorderStyle   = 1
    DragMode      = 0
    Enabled       = -1
    ForeColor     = QBColor(0)
    Height        = Char(5)
    Left          = Char(16)
    MousePointer  = 0
    MultiLine     = 0
    ScrollBars    = 0
    TabIndex      = 2
    TabStop       = -1
    Tag           = ""
    Text          = "Text1"
    Top           = Char(2)
    Visible       = -1
    Width         = Char(35)
END
BEGIN OptionButton Option1
    BackColor     = QBColor(7)
    Caption       = "英語"
    DragMode      = 0
    Enabled       = -1
    ForeColor     = QBColor(0)
    Height        = Char(3)
    Left          = Char(0)
```

```
    MousePointer = 0
    TabIndex     = 3
    TabStop      = -1
    Tag          = ""
    Top          = Char(1)
    Value        = -1
    Visible      = -1
    Width        = Char(15)
END
BEGIN OptionButton Option2
    BackColor    = QBColor(7)
    Caption      = "日本語"
    DragMode     = 0
    Enabled      = -1
    ForeColor    = QBColor(0)
    Height       = Char(3)
    Left         = Char(0)
    MousePointer = 0
    TabIndex     = 4
    TabStop      = 0
    Tag          = ""
    Top          = Char(3)
    Value        = 0
    Visible      = -1
    Width        = Char(15)
END
BEGIN OptionButton Option3
    BackColor    = QBColor(7)
    Caption      = "フランス語"
    DragMode     = 0
    Enabled      = -1
    ForeColor    = QBColor(0)
    Height       = Char(3)
    Left         = Char(0)
    MousePointer = 0
    TabIndex     = 5
    TabStop      = 0
    Tag          = ""
    Top          = Char(5)
    Value        = 0
    Visible      = -1
    Width        = Char(15)
END
BEGIN OptionButton Option4
    BackColor    = QBColor(7)
    Caption      = "ドイツ語"
    DragMode     = 0
    Enabled      = -1
    ForeColor    = QBColor(0)
    Height       = Char(3)
    Left         = Char(0)
    MousePointer = 0
    TabIndex     = 6
    TabStop      = 0
    Tag          = ""
    Top          = Char(7)
```

```
        Value          = 0
        Visible        = -1
        Width          = Char(15)
    END
END

SUB Command1_Click ()
    TRUE = -1
    IF Option1.value = TRUE THEN
        Text1.Text = "Hello, world"
    ELSEIF Option2.value = TRUE THEN
        Text1.Text = "世界のみなさん、こんにちは"
    ELSEIF Option3.value = TRUE THEN
        Text1.Text = "Bonjour tout le monde"
    ELSE
        Text1.Text = "Hallo, welt"
    END IF
END SUB

SUB Command2_Click ()
    SCREEN.HIDE
    END
END SUB
```

A.3 ASCII フォームのロードと保管

ASCII フォームは、バイナリ形式で保存されたフォームより、Visual Basic への読み込みに時間がかかります。これは、ASCII フォームをメモリへ読み込む前に、いったんバイナリ形式へ変換しているためです。もし、ASCII フォーム中でフォーム情報に対してコメントを記述していても、この段階で失われます。

ASCII フォームを Visual Basic へロードして、変更を加えて保管すると、もとのファイルが上書きされます。

ファイルのロード時のエラー

ASCII フォームのロード時にエラーが発生すると、ログ ファイルにエラーが記録されます。エラーが発生するたびに、ログ ファイルにエラーが追加されます。ロードが終了すると、エラーを記録したログ ファイルが作られたことを示すエラーメッセージが表示されます。ログ ファイルの名前は、ASCII フォームの拡張子を LOG に変えたものです。例えば、MYFORM.FRM のログ ファイルは MYFORM.LOG となります。

エラーの種類

ASCII フォームのロード中に発生するエラーには、次の 2 種類があります。

- フォームのロードを続行可能な致命的でないエラー
- フォームのロードを続行できない致命的エラー

ASCII フォームのロード時に発生する可能性のあるエラーを、次に示します。

エラー	エラーの種類	内容
Syntax errors	非致命的	プロパティの名前やコントロールの種類を示す名前に問題があります。プロパティ名が存在しない場合は、そのプロパティは無視され、エラーメッセージがログ ファイルに残されます。コントロールの種類が存在しない場合は、同じ名前を持つフレームに変換されます。
Unexpected End of File error	致命的	END の前に、ファイルの終わりに到達しました。
Property Value errors	致命的	プロパティの値が不適切で、デフォルトの値で代用することができません。Index, CtlName, FormName などのプロパティに同じ値を複数指定したり、不適切な値を指定すると、このエラーが発生します。
Property Value errors	非致命的	プロパティの値が不適切だったのでデフォルトの値で代用しました。デフォルトの値とは、フォーム デザイナで使用されるものと同じです。
No Version Number	非致命的	バージョン番号がないか、ファイルの 1 行目にありません。ファイルは、Visual Basic for MS-DOS のフォームとしてロードされます。
Missing first Begin	致命的	バージョン番号の後に、BEGIN がありません。

B メモリ管理

ここでは、Visual Basic for MS-DOS でメモリを効率的に利用するための方法を説明します。プログラミング環境は、XMS メモリと EMS メモリの両方を利用できます。作成したアプリケーションをコンパイルして .EXE ファイルを作成すると実行速度が上がりますが、プログラミング環境上での実行速度や効率を高めるためにも、いくつかの方法があります。また、アプリケーションにもっとも適したコードを作成できるように、Visual Basic for MS-DOS における変数とオブジェクトのメモリへの格納方法についても説明します。さらに、メモリが不足しているときにアプリケーションを実行する場合の対策についても触れます。

B.1 利用可能メモリの設定

プログラミング環境上で実行できるアプリケーションのサイズは、プログラミング環境自身が使うメモリ量によって決まります。プログラミング環境がコンベンショナルメモリ (メインの 640 KB、一部の機種では 768 KB) により多くの空き領域を残せば、それだけ大きなアプリケーションが実行できることになります。

Visual Basic for MS-DOS はオーバーレイを使って、プログラミング環境自身の一部をコンベンショナル メモリの外側に追い出します。Visual Basic for MS-DOS は、XMS メモリ，EMS メモリ，ディスクの順に利用可能かどうかを調べ、利用可能であれば自身の一部をその領域に退避させます。ここでは、プログラミング環境のサイズを設定する方法と、XMS メモリや EMS メモリを利用する方法について説明します。

プログラミング環境のサイズ

プログラミング環境がコンベンショナル メモリに占めるサイズを調節するには、VBDOS.EXE の /S:n コマンドライン オプションを使います。/S オプションで 1 を指定すると、アプリケーションが使用できるメモリ量が最大になります。

```
VBDOS /S:1 myprog.mak
```

この例を実行すると、アプリケーションに対して最大限のメモリを残すように、メモリの利用方法が調節されます。これは同時に、プログラミング環境が頻繁にスワップを行うことを意味するため、プログラミング環境の処理スピードは低下します。/S オプションで 640 を指定すると、プログラミング環境はすべてコンベンショナル メモリにロードされ、スワップは行われなくなりますが、アプリケーションが使用できるメモリ量は最も少なくなります。

```
VBDOS /S:640 myprog.mak
```

プログラミング環境のコードのうち、どの部分がスワップされるかは、自動的に判断されます。性能に大きな影響を与えない部分が優先的にスワップされ、例えばコードリストの出力を行う部分はスワップされますが、デバッグ時にシングル ステップでアプリケーションを実行するコードはスワップされません。

Visual Basic for MS-DOS の性能を最大限に引き出すためには、/S オプションには適切な値を指定する必要があります。例えば、小さなアプリケーションを実行するときに、アプリケーション用に多くのコンベンショナル メモリを確保しても意味がありません。

▼ **最適な設定を見つけるには**

1. 最初はデフォルトの設定 (/S スイッチを指定しない) で実行します。
2. メモリが不足するまで、プログラミング環境で作業します。
3. メモリが足りなくなったらいったんプロジェクトを保管し、Visual Basic for MS-DOS を終了します。
4. /S で中程度の値 (300 など) を指定して、Visual Basic for MS-DOS を再起動します。
5. /S で指定する値を 50 ずつ小さくして、ステップ 2 以降の手順を繰り返します。

注　意　/S オプションで指定する値は相対的なものです。/S:1 は Visual Basic for MS-DOS が使うサイズが最小になることを指定しますが、プログラミング環境のサイズが 1 KB になるわけではありません。同じように /S:640 は、プログラミング環境のサイズが 640 KB になることを指定するわけではありません。これはプログラミング環境の実際のサイズが 640 KB より小さいことからも分かります。

XMS メモリと EMS メモリの使い方

プログラミング環境が使う XMS メモリと EMS メモリのサイズは、/X スイッチと /E スイッチで指定します。これらのスイッチを使うと、プログラミング環境とアプリケーションで EMS メモリを共有したり、これらのメモリをすべて空けて、Visual Basic for MS-DOS を強制的にディスクにスワップするように設定することができます。

例えば次のコマンド ラインでは、/S オプションでプログラミング環境のサイズを設定し、EMS メモリをアプリケーション用に 100K、プログラミング環境用に 200K 確保し、XMS メモリは使わない (0K) ことを指定しています。

```
VBDOS /S:250 /E:100,200 /X:0
```

▼ **MS-DOS バージョン 5.0 で空きメモリの容量を知るには**

- MS-DOS バージョン 5.0 の **mem /c** コマンドを実行します。

mem コマンドは、プログラムごとのメモリの使用状況を画面に表示します。またコンベンショナル メモリ，EMS メモリ，XMS メモリの空き容量をバイト単位で示します。/E スイッチと /X スイッチでは、サイズをキロバイト単位で指定するので、表示される値を 1024 で割る必要があります。

MS-DOS バージョン 5.0 を使っていないときでも、Visual Basic for MS-DOS を使って EMS メモリの利用可能サイズを調べることができます。

▼ EMS メモリの空きサイズを決定するには

1. /E スイッチで 0 を指定して、Visual Basic for MS-DOS を起動します。この場合、プログラミング環境用に EMS メモリが割り当てられることはありません。

```
VBDOS /E:,0
```

2. 次のコードを入力して実行します。

```
PRINT FRE(-3)
```

EMS メモリの利用可能サイズがキロバイト単位で示されます。この値を上限として、/E スイッチを使って、アプリケーションとプログラミング環境で共有する EMS メモリのサイズを指定します。

XMS メモリや EMS メモリを利用可能にするには、次のようなメモリを管理するためのソフトウェアが必要です。

- XMS メモリ: Microsoft Windows や MS-DOS で提供されている HIMEM.SYS などの XMS ドライバを使って設定します。
- EMS メモリ エミュレータ: ソフトウェアEMS と呼ばれる方法で、エクステンデッド メモリの一部またはすべてを EMS メモリとしてエミュレートします。XMS メモリ ドライバの多くは、この機能をサポートしています。Microsoft Windows と MS-DOS では、EMS メモリ エミュレータとして EMM386.EXE や EMM386.SYS が提供されています。
- EMS メモリ: EMS メモリ ボードを装着している場合は、そのボードをコントロールする EMS メモリ ドライバを使って設定します。

Visual Basic for MS-DOS は、XMS メモリと EMS メモリの両方を同時に使うことができます。

XMS メモリの使い方

XMS メモリを使うには、80286 以降の CPU が必要です。Visual Basic for MS-DOS で XMS メモリを利用するには、XMS メモリ ドライバが Extended Memory Specification (XMS) バージョン 2.0 以降に準拠している必要があります。

XMS メモリが利用可能なとき、Visual Basic for MS-DOS はプログラミング環境自身を可能な限り XMS メモリに格納します。プログラミング環境が利用する XMS メモリの量を制限するには、/X:n スイッチ (n は XMS メモリにスワップするプログラミング環境のキロバイト数) を使います。

EMS メモリの使い方

EMS メモリは 8086 系のすべての CPU で使用可能です。Visual Basic for MS-DOS で EMS メモリを利用するには、EMS ドライバが Lotus-Intel-Microsoft (LIM) Expanded Memory Specification (EMS) バージョン 4.0 に準拠している必要があります。EMS メモリを使うには、CONFIG.SYS ファイルで EMS メモリ ドライバを設定してください。それぞれの EMS メモリ ボードについて、対応するデバイス ドライバが提供されています。

EMS ドライバは、16K バイトを 1 ページとして、コンベンショナル メモリの内容を 8086 のアドレス可能領域にスワップします。このため、モジュールや配列を EMS メモリに格納するには、最大 16 KBのサイズに収める必要があります。

EMS メモリが利用可能なとき、Visual Basic for MS-DOS はプログラミング環境自身を可能な限り EMS メモリに格納します。プログラミング環境が利用する EMS メモリの量を制限するには、E:*n*,*m* スイッチを使います。*n* はアプリケーション用に確保するサイズを示し、*m* はプログラミング環境自身のために確保するサイズを示します。どちらも KB 単位で指定してください。

RAM ディスクの使い方

テンポラリ ファイルを使うユーティリティ (LINK など) では、環境変数 TMP を RAM ディスク上のディレクトリに設定すると、テンポラリ ファイルは RAM ディスク上に作成され、処理が高速化します。また、ライブラリや頻繁にアクセスする実行可能ファイルをあらかじめ RAM ディスク上にコピーしておくと、実行速度が上がります。何らかの理由でハングアップしたときにシステムを再起動すると、RAM ディスク上の情報は失われるため、ソース ファイルを RAM ディスクに置いて開発することは危険です。

ディスク キャッシュの使い方

ディスク キャッシュ ユーティリティを使うと、ハード ディスクに対して最後にアクセスしたデータがメモリの一部に一時的に保管されます。必要な情報がキャッシュにあるときは、データアクセス時間が大幅に短縮します。

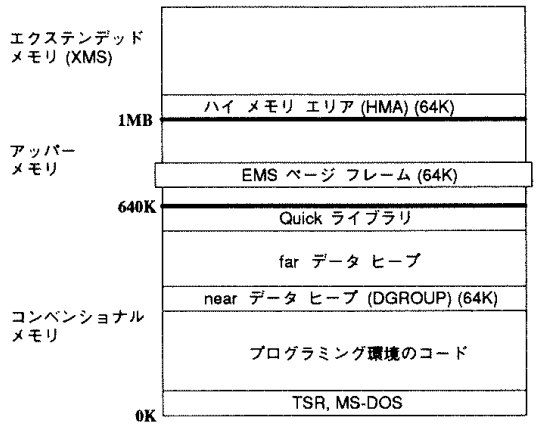
MS-DOS バージョン 5.0 や Microsoft Windows では、ディスク キャッシュ ユーティリティとして SMARTDRV.SYS または SMARTDRV.EXE が提供されています。

B.2 Visual Basic for MS-DOS のメモリの使い方

Visual Basic for MS-DOS では、アプリケーションを実行するときに、そのコードとデータをメモリ (RAM) 上に格納します。プログラミング環境でアプリケーションを実行するときと、コンパイル済みの実行可能 (.EXE) ファイルの状態でアプリケーションを実行するときとでは、メモリの使い方が異なります。

プログラミング環境のメモリ マップ

次のメモリ マップは、プログラミング環境でアプリケーションを実行するときに、そのコードとデータがメモリに格納されるようすを示しています。



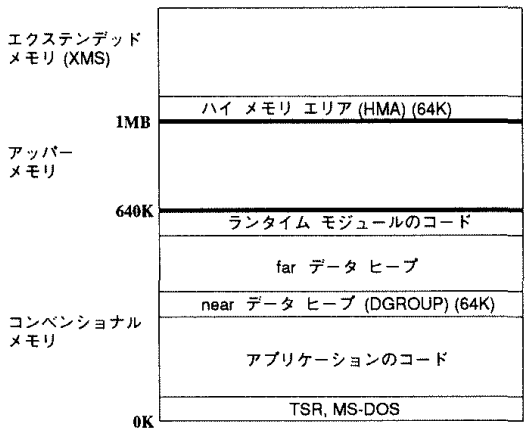
メモリ領域	説明
XMS メモリ 1 MB より上位のメモリ	適切な XMS ドライバによって有効になっているとき、プログラミング環境自身の一部が退避します。
EMS メモリ EMS ドライバで制御される 16K 単位に切り分けられた領域	適切な EMS ドライバによって有効になっているとき、プログラミング環境自身の一部，アプリケーションに含まれる 16K 未満のプロシージャとモジュールレベル コードが格納されます。それぞれ 16K を超える場合は、far ヒープに置かれます。
ハイ メモリ エリア (HMA) メインの 1 MB のすぐ上位の位置にある約 64K のメモリ	Visual Basic for MS-DOS では直接使いませんが、MS-DOS バージョン 5.0 を使うときは、MS-DOS の一部をこの領域にロードして、プログラミング環境用にメモリを解放します。
アッパー メモリ (UMB) 640 KB と 1 MB の間のメモリ	Visual Basic for MS-DOS で EMS メモリを使うときは、この領域に 64K のページ フレーム ブロックを割り当てる必要があります。 EMS のページ フレームは 4 つの 16K ブロックに分けられます。
Quick ライブラリの格納位置 コンベンショナル メモリの最上位バイトから下位へ	プログラミング環境で Visual Basic for MS-DOS の Quick ライブラリをロードすると、この領域に格納されます。Quick ライブラリをロードしないときは、コンベンショナル メモリの最上位まで far データ ヒープとして使用されます。 Quick ライブラリは EMS メモリには格納できません。

far データ ヒープ	far データ ヒープには、次の項目が含まれます。 ■ EMS メモリ中のコードを除く、開発中のアプリケーションを構成するすべてのコード。 ■ モジュール レベルの名前付き COMMON ブロックに含まれる可変長文字列と文字列配列の要素は、最大 64K の個別のセグメントに格納されます。 ■ プロシージャレベルの可変長文字列とテンポラリ文字列は、最大 64K の個別のセグメントに格納されます。 ■ 名前なし COMMON ブロックに含まれる可変長文字列は、最大 64K の個別のセグメントに格納されます。 ■ プロシージャ レベルの可変長文字列配列は、プロシージャごとに、最大 64K の個別のセグメントに格納されます。 ■ COMMON ステートメントで先に宣言された ものを除く、すべての配列データ。 ■ 静的なフォーム定義。それぞれのフォーム定義につき最大 64K です。 ■ ロードされているフォームの動的な操作用のコピー。それぞれのコピーにつき最大 64K です。 ■ プログラミング環境の内部データ構造。 ■ ファイル データ ブロック。 ■ オンライン ヘルプ バッファ。
near データ ヒープ (DGROUP)	near データ ヒープには、次の項目が含まれます。 ■ 文字列記述子，配列記述子，オブジェクト記述子。 ■ 単純変数。 ■ プログラミング環境の内部データ構造。 ■ アプリケーションのスタック。 ■ Quick ライブラリがロードされているときは、それに含まれる単純変数，静的配列データ，COMMON データ。 ■ プログラミング環境の内部変数。
プログラミング環境のコード	XMS または EMS メモリ，またはディスク (XMS または EMS がないとき) に退避されている部分を除く、プログラミング環境のコード。
メモリ常駐 (TSR) プログラムと MS-DOS	プログラミング環境を起動する前にロードされていた TSR プログラム (MS-DOS を含みます)。MS-DOS はハイ メモリ エリア (HMA) にロードされる場合もあります。

注 意 プログラミング環境では、それぞれのプロシージャおよびモジュールレベルコードのセグメントは、最大 64K まで可能です。コンパイルして作成した .EXE ファイルでは、モジュール全体で 64K 以内である必要があります。プログラミング環境では、64K を超えるモジュールを実行できます。

実行可能 (.EXE) ファイルのメモリ マップ

次のメモリ マップは、Visual Basic for MS-DOS で作成した実行可能 (.EXE) ファイルを実行するときに、そのコードとデータがメモリに格納されるようすを示します。



次の表は、このメモリ マップの各部分について説明します。

メモリ領域	説明
XMS メモリ 1 MB より上位のメモリ	使用しません。
EMS メモリ EMS ドライバで制御される 16K 単位に切り分けられた領域	使用しません。
ハイ メモリ エリア (HMA) メインの 1 MB のすぐ上位の位置にある約 64K のメモリ	使用しません。
アッパー メモリ (UMB) 640 KB と 1 MB の間のメモリ	使用しません。
ランタイム モジュールのコード コンベンショナル メモリの最上位バイトから下位に	分離型の実行可能ファイルでだけ使います。独立型の実行可能ファイルでは、この領域はありません。

far データ ヒープ	far データ ヒープには、次の項目が含まれます。 ■ モジュール レベルの名前付き COMMON ブロックに含まれる可変長文字列と文字列配列の要素は、64K の個別のセグメントに格納されます。 ■ プロシージャレベルの可変長文字列は、 64K の個別のセグメントに格納されます。 ■ 名前なし COMMON ブロックに含まれる可変長文字列は、64K の個別のセグメントに格納されます。 ■ プロシージャ レベルの可変長文字列配列は、プロシージャごとに、64K の個別のセグメントに格納されます。 ■ すべての動的数値および固定長文字列配列のデータ。 ■ 静的なフォーム定義。それぞれのフォーム定義につき最大 64K です。 ■ ロードされているフォームの動的な操作用のコピー。それぞれのコピーにつき最大 64K です。 ■ リテラル文字列。 ■ ファイル データ ブロック。
near データ ヒープ (DGROUP)	near データ ヒープには、次の項目が含まれます。 ■ 文字列記述子，配列記述子，オブジェクト記述子。 ■ すべての単純変数 (常に far データ ヒープに格納される可変長文字列データを除きます)。 ■ すべての静的配列データ (可変長文字列データを除きます)。 ■ 記号定数の値。 ■ COMMON データ (可変長文字列データを除きます)。 ■ アプリケーションのスタック。 ■ Visual Basic for MS-DOS ランタイムの内部変数。
アプリケーションのコード	アプリケーションのコード セグメント。コード セグメントは各モジュールごとに最大 64K です。
メモリ常駐 (TSR) プログラムと MS-DOS	アプリケーションを起動する前にロードされていた TSR プログラム (MS-DOS を含みます)。MS-DOS はハイ メモリ エリア (HMA) にロードされる場合もあります。

B.3 データとオブジェクトのサイズ

効率よくメモリを使うには、Visual Basic for MS-DOS のデータとオブジェクトの格納方法について理解しておく必要があります。例えば整数は、単精度および倍精度の浮動小数点数よりも少ないメモリで格納できます。フォームとコントロールは、種類によってメモリに占める量が異なります。これらのメモリ割り当ての方法を知ることによって、プログラムの効率を高めることができます。

データ型

Visual Basic for MS-DOS は、データをさまざまな方法で格納します。データを扱うとき、そのデータを格納可能な最小サイズのデータ型を選択すれば、メモリを節約し、プログラムの効率を高めることができます。

次の表は、それぞれのデータ型で使うメモリ量を示します。

データ型	DGROUP のバイト数
整数	2
長整数	4
単精度浮動小数点数	4
倍精度浮動小数点数	8
通貨型数	8
可変長文字列	最大 32,767 バイト
固定長文字列	最大 32,767 バイト

注　意　配列を扱う場合、配列に含まれる個々の要素のデータの他に、“配列記述子”を使って必要な情報を保持しています。配列記述子は、配列に含まれる実際のデータが格納されているアドレスやサイズを示しています。したがって、配列の格納に必要なメモリ量は、個々の要素のメモリ量を合計したものより多くなります。また、文字列変数を格納するときは、4 バイトの“文字列記述子”が必要です。このように、本来のデータ以外に必要な部分を、オーバーヘッドと呼びます。

プログラミング環境上での実行か .EXE ファイルの実行かにかかわらず、各変数には DGROUP 内のメモリが割り当てられます。プログラミング環境では、モジュール レベルの単純変数に割り当てられる合計サイズが、各モジュールにつき 32K を超えることができません。

ユーザー定義型のオーバーヘッド

プログラミング環境では、ユーザー定義型の変数は、そのデータ型の各要素に必要なメモリとオーバーヘッドを合計したサイズの領域を必要とします。次に例を示します。

```
TYPE PowerOfTwo
    I AS INTEGER            'データ 2 バイト, オーバーヘッド 8 バイト
    L AS LONG               'データ 4 バイト, オーバーヘッド 8 バイト
    S AS SINGLE             'データ 4 バイト, オーバーヘッド 8 バイト
    D AS DOUBLE             'データ 8 バイト, オーバーヘッド 8 バイト
    St AS STRING * n        'データ n バイト, オーバーヘッド 4 バイト
END TYPE
```

非実行可能ステートメントのオーバーヘッド

非実行可能ステートメント (**COMMON**, **DECLARE**, **CONST**) のオーバーヘッドは、プログラミング環境上で実行するか、.EXE ファイルとして実行するかによって異なります。

プログラミング環境	.EXE ファイル
COMMON 変数ごとに 8 バイトと、その変数の格納に必要なバイト数	**COMMON** オーバーヘッドなし
DECLARE それぞれのプロシージャ (**SUB** または **FUNCTION**) につき 46 バイト。プロシージャを定義する前に宣言しているときは、宣言時に 46 バイトのオーバーヘッドが作成されます。プロシージャを定義しないときでも、46 バイトはそのまま割り当てられます。	**DECLARE** オーバーヘッドなし
CONST オーバーヘッドなし。定数は変数と同じように扱われます (ただし読み込み専用のフラグが付きます)。	**CONST** オーバーヘッドなし。定数は変数と同じように扱われます。

配列と文字列のオーバーヘッド

それぞれの配列は、データ型やメモリ割り当ての方法 (動的または静的) にかかわらず配列記述子を持ちます。各配列記述子は、DGROUP 内の 14 バイトと、配列の各次元ごとに 4 バイトを使います。例えば 3 次元配列は、データ型にかかわらず配列記述子として 26 バイトを使い、さらに配列のデータ型に応じたバイト数が必要になります。

可変長文字列は、DGROUP 内の文字列記述子 4 バイトと、far ヒープ内の文字列データに追加される 6 バイトをオーバーヘッドとして使い、さらに 1 文字につき 1 バイトを必要とします。

可変長文字列配列には、一般的な配列記述子の他に、各要素を記述するための文字列記述子の配列が含まれます。1 つの可変長文字列配列は、記述子として 14 バイト，配列の各次元につき 4 バイトを使います。例えば、3 次元の可変長文字列配列で、それぞ

れの次元に 10 個の文字列が含まれるとき、オーバーヘッドは (14 + 3 * 4 + 3 * 10 * (4 + 6)) バイトになります。

次の表は、データ，配列記述子，文字列記述子が DGROUP と far データ ヒープにどのように格納されるかを示します。

変数のデータ型	DGROUP (near データ ヒープ)	far データ ヒープ
単純変数 (数値と固定長文字列)	データ	(なし)
単純可変長文字列変数	文字列記述子	データ + 文字列あたり 6 バイト
静的配列変数 (数値と固定長文字列)	プログラミング環境上: データ EXE ファイル: 配列記述子	プログラミング環境上: (なし) EXE ファイル: データ
動的配列変数 (数値と固定長文字列)	配列記述子	データ
静的または動的可変長文字列配列	配列記述子 文字列記述子	データ + 文字列あたり 6 バイト

注 意 プログラミング環境では、変数名は DGROUP ではなく far データ ヒープに格納されます。それぞれの変数名は、オーバーヘッド 4 バイトと、その名前をテキストとして格納するのに必要なバイト数を使います。

フォーム，コントロール，モジュールのオーバーヘッド

プログラミング環境では、各モジュールは DGROUP のうち約 150～250 バイトを使います。

フォームとコントロールは、オーバーヘッドとして数バイト (4～14 バイト) を使いますが、このサイズはすでに参照されているかどうかと、プログラミング環境上での実行か .EXE ファイルとしての実行かによって異なります。それぞれのフォームとコントロールが占めるメモリ サイズは、そのプロパティの設定値によって大きく変化するので、フォームとコントロールが使うメモリ サイズを知りたいときは、**FRE** 関数を使って実際のサイズを調べてください。例えば次のコードは、プログラムの Form1 で使われている動的な領域のサイズを示します。

```
SUB Form_Click ()
    Loaded! = FRE(-1)
    UNLOAD Form1
    Unloaded! = FRE(-1)
    MSGBOX STR$(Unloaded! - Loaded!)
END SUB
```

注　意　FRE(-4) を使って、ActiveForm プロパティで参照できるカレント フォームのデータ セグメントの空き領域のサイズを調べることもできます。このセグメントには、フォームのすべてのプロパティとコントロールが含まれます。

LOAD ステートメントによるコントロール配列への要素の追加、リスト ボックスへの項目の追加、文字列が設定できるプロパティへの長い文字列の割り当てなどの操作を行うと、実行時にフォームが使うメモリ サイズが増えます。また、**UNLOAD** ステートメントでフォームをアンロードすると、そのフォームが占めていたメモリの一部は解放されますが、far データ ヒープ中の一部の領域は確保されたまま残ります。呼び出しシーケンスをさかのぼり、そのフォームが最初にロードされた時点までスタックが戻ると、このメモリ領域は解放されます。

C プログラム作成上のヒント

ここでは、より高速で効率のよいプログラムを作成するためのヒントについて説明します。例えば、速度とサイズのどちらを優先するかによって、プログラミングのときに考えるべき点が異なります。

C.1 サイズを小さくするためのヒント

データ領域を確保したいとき、次のようなコーディングのスタイルが役に立ちます。

- プログラムを作成する際には、できる限り変数の代わりに定数を使ってください。Visual Basic プログラム中の通常の数値変数は、DGROUP に置かれます。一方、コンパイルされたプログラムでは、定数は生成されたコード中に直接埋め込まれるため、これらの定数を保持するために DGROUP は使われません。もし変数の値がプログラムの実行中に変化しないのであれば、定数に置き換えることによって、プログラムを高速化できると同時に、メモリの使用量も少なくなります。
- **SCREEN** ステートメントの引数は定数で指定してください。**SCREEN** ステートメントの引数に変数を使うと、コンパイル時にはどのスクリーン モードになるか判断ができないため、すべてのスクリーン モードに対応できるように、大量のコードが実行可能 (.EXE) ファイルに組み込まれてしまいます。定数で指定すれば、そのスクリーン モードに対応したコードだけが組み込まれます。また、グラフィックスを使わないプログラムでは、テキスト モードに設定してください。
- **SUB** プロシージャや **FUNCTION** プロシージャの内部では、できるだけローカル変数を使ってください。プロシージャ内部のローカル変数は、そのプロシージャが実行されているときだけ DGROUP を占有します。Visual Basic では、ローカル変数はスタックに置かれます。ローカル変数を使うプロシージャの実行が終わると、スタックに置かれたローカル変数は破棄され、スタックは元の状態に戻ります。静的変数，共有されている変数，モジュール レベルで使われている変数などは、ローカル変数と異なり、プログラムの実行中は常に DGROUP を占有します。
- ファイル I/O バッファと文字列領域のバランスに注意してください。ファイル I/O バッファは、可変長文字列と同じ領域に確保されます。ファイル I/O バッファのサイズは、**OPEN** ステートメントの **LEN =** 引数で指定できます。デフォルトでは、シーケンシャル アクセスの I/O バッファのサイズは 512 バイト、ランダム アクセスでは 128 バイトです。一般に、I/O バッファを大きくするとディスク アクセスの回数が減り、プログラムは高速になります。一方、バッファを小さくすると、far ヒープの消費量が少なくなります。

- できるだけプロシージャを使うようにします。繰り返し行う処理をプロシージャにしておくと、コードを小さくできるだけでなく、読みやすく、保守やデバッグが楽なプログラムにすることができます。また、汎用性のあるプロシージャにしておけば、他のプログラムで簡単に再利用できます。
- できるだけ整数の変数を使うようにします。Visual Basic では、特に宣言しない場合のデフォルトのデータ型は、単精度浮動小数点型です。整数が 2 バイトで格納できるのに対し、単精度浮動小数点型は 4 バイトを使います。整数で済む場合は、**DIM** ステートメントで明確に宣言するか、変数名の最後に型宣言文字 % を付けて、整数型であることを示してください。

注　意　付録 B「メモリ管理」には、Visual Basic がどのようにメモリを管理しているかが説明されています。プログラムやデータがメモリ中にどのように配置されるかを知っておけば、プログラムの効率を上げるためにより適切な判断ができます。

C.2　高速化するためのヒント

プログラムの実行スピード、特に実行可能 (.EXE) ファイルの実行スピードを上げるためには、次のような点を考慮してください。

高速化のためのデータの扱い

変数のデータ型や静的/動的の選択は、プログラムの実行スピードに対して大きな影響があります。適切な選択を行うだけで、高速化が可能です。

定数

メモリを節約するとともに、高速化を図るには、変数の代わりにできるだけ定数を使うようにしてください。記号定数はコンパイル時に実際の数値または文字列に変換され、生成されるコード中にはその値が直接埋め込まれます。定数に置き換えができるときは、ほとんどの場合、変数より効率的です。

整数

整数 (**INTEGER**) は、Visual Basic では最も高速なデータ型です。整数は、他のどの数値データ型よりも小さく (1 つの整数型のデータにつき 2 バイト)、コンパイラが生成したコードからランタイム呼び出しを行う必要もないために効率的です。整数型の型宣言文字は % です。1 つのプログラミングのスタイルとして、各モジュールとプロシージャの先頭で常に DEFINT A-Z を宣言し、必要なときに限って整数以外のデータ型を宣言する方法もあります。

プログラムの高速化のために整数型を活用するヒントには、次のようなものがあります。

- ループのカウンタ変数は常に整数にする。

  ループ カウンタ変数を使うループでは、変数が整数型であるか確認してください。特に指定しないと、デフォルトのデータ型である単精度浮動小数点数 (**SINGLE**) になり、ループの実行速度が低下するとともに、プログラムのサイズが増大します。特にほかに浮動小数点数を使っていない場合は、実行可能 (.EXE) ファイルを作成するときにこの変数のためだけに浮動小数点演算パッケージがリンクされてしまいます。

- 論理値は整数型にする。

  ループ カウンタと同様に、オン/オフや限定された選択肢のうちの 1 つを表すときは、整数型が適当です。

- ステートメントや関数への引数は、できる限り整数にする。

  特に別のデータ型にする必要がなければ、必ず整数型を使うようにします。

通貨型

Visual Basic では、固定小数点数である通貨型 (**CURRENCY**) を利用できます。通貨型の型宣言文字は @ です。通貨型は、小数点以下の桁が表せるようにシフトされた、非常に長い整数と考えることができます。通貨型は、内部的には 8 バイトの整数であり、実際の値はそれを 10,000 で割ったものとなります。これによって、小数点以下 4 桁、小数点以上 15 桁の数値を表しています。内部的には整数として表現されているので、加減算では浮動小数点数より速く、その他の演算では逆に遅くなります。

2 進化 10 進数 (BCD) と比べると、BCD が 1 バイトで表せる 256 の値の内の 100 しか使っていないため、通貨型は、同じバイト数で比較して BCD よりはるかに大きな表現力を持っていると言えます。また、加減算のような数値演算についても、通貨型の方が高速です。

静的配列と動的配列

配列を静的にするか動的にするかは、容量と速度のどちらを優先するかによって決めてください。静的配列は動的配列よりも速く参照できますが、動的配列は far メモリに格納されるため、よりサイズの大きな配列が作成できます。同様に、標準的な動的配列にアクセスするのは、/Ah オプションを付けてコンパイルする huge 配列にアクセスするよりも高速です。

高速化のための制御構造の最適化

Visual Basic は、**GOTO** ステートメント、**GOSUB / DEF FN** ステートメントと、**SUB / FUNCTION** プロシージャという、3 レベルの汎用的な制御構造をサポートしています。コード中で制御を分岐するときにかかる時間は、一般的に、分岐先に渡される情報の量と、そのときの状況として保持しておかなければならない情報の量に依存します。

GOTO ステートメント

GOTO ステートメントは、Visual Basic の分岐制御の中で、最も古く、最も構造化されていません。しかし、**GOTO** ステートメントはアセンブリ言語の **JMP** 命令に直接変換されるので、プログラム中の A 地点から B 地点へ分岐するような単純なケースでは最も高速な方法です。構造化プログラミングの観点からは、**GOTO** ステートメントを多用すると、プログラムの可読性やデバッグのしやすさ、保守性などが低くなるため、おすすめできるプログラミング スタイルではありません。

GOSUB ステートメントと DEF FN サブルーチン

GOSUB および **DEF FN** ステートメントは、**GOTO** と同様にあまり構造化されていませんが、サブルーチンまたは **DEF FN** ブロックの実行終了後に制御が元の位置に戻る点が異なります。内部的には、戻り先のアドレスをスタックに積み、処理の終了後にリターン命令を実行して、アセンブリ言語の **CALL** 命令の直後の命令に戻っています。

SUB および FUNCTION プロシージャの呼び出し

SUB および **FUNCTION** プロシージャは構造化された強力な制御構文で、パラメータの受け渡し，ローカル変数，再帰的な呼び出し，ローカルなエラー処理，モジュール間呼び出し，複数言語の使用などが可能になっています。このようにプロシージャは非常に強力ですが、それだけ内部的な処理も多く、制御の受け渡しには多少のオーバーヘッドがあります。特定の状況では、GOTO や GOSUB ステートメントの方が高速です。

プロシージャの呼び出しに必要な時間は、呼び出しの前にスタックに保存する、そのときの状況を示す情報とパラメータ情報 (スタック フレームと呼ばれます) の量に依存します。スタック フレームの大きさは、プロシージャに渡されるパラメータのデータ型や数，エラーをトラップするための情報，プロシージャに含まれるコード自体などによって決まります。

呼び出しの速度を向上するためのヒントを、次に示します。

- パラメータの数を少なくします。一般的には、グローバル変数を使うよりパラメータを使う方が好ましいのですが、パラメータをスタックに積むための時間が必要になります。呼び出しの速度が問題になるような場合には、グローバル変数を使うと、スタック フレームのサイズが減り、プロシージャを呼び出すときの時間を短縮できます。
- 値による引き渡しを使います。動的配列の要素や固定長文字列変数を引数として渡すときには、カッコで囲むと式と見なされて参照ではなく値で渡されるため、呼び出しの時間が短くなります。また、**DECLARE** ステートメントで **BYVAL** キーワードを指定しても、値で渡されます。もちろん、パラメータとして指定した変数に対する変更を呼び出し側でも有効にしたい場合には、参照による受け渡しを行わなければなりません。

- **SUB** または **FUNCTION** ステートメントで、**STATIC** キーワードを指定します。このようにすると、プロシージャ中の変数は自動的に静的 (STATIC) 変数となり、プロシージャの呼び出しの前後においても存在するので、呼び出しの度に再生成されることはありません。ただしこの方法を使うと、変数の分だけ多くの領域が必要です。

ループの高速化

ループは単純な処理ですが、高速化を図る方法がいくつかあります。第 1 に、前にも説明したとおりループ カウンタ変数には整数を使ってください。第 2 に、繰り返しのたびに同じ処理を行うコードをループの外に移してください。例えば、次のようなループは効率的ではありません。

```
FOR I%=1 to LastLoop%
    A(I%)=SIN(B)
NEXT I%
```

より高速になるように書き直すと、次のようになります。

```
C=SIN(B)
FOR I%=1 to LastLoop%
    A(I%)=C
NEXT I%
```

入出力の速度の改善

ファイルや画面に対する入出力は、しばしば Visual Basic アプリケーションの速度的なボトルネックになります。これは、ディスクへのアクセスや画面への書き込みが比較的遅い処理だからです。ここでも、他の部分と同様に、実行速度を向上させるためのいくつかのポイントがあります。

シーケンシャル アクセス

OPEN ステートメントで **LEN=** 引数を指定すると、シーケンシャル アクセスで使う入出力バッファの大きさを設定できます。デフォルトのサイズは 512 バイトです。このバッファは、far ヒープに置かれます。バッファが大きいほど、ディスクへのアクセス回数が減り、処理が高速化されます。ただし大きなバッファを確保すると far メモリを消費するため、可変長文字列用の領域がそれだけ小さくなります。

画面への表示

文字列の画面への表示は、**PRINT** ステートメントを適切に使うことによってある程度高速化できます。**PRINT** ステートメントは、内部で多くのルーチンを呼び出し、特に最後にセミコロンが指定されていない場合には自動スクロールを行います。改行する必要がなければ、**PRINT** ステートメントの最後にセミコロンを付けてください。これによって、低レベルのプリント ルーチンが不必要に呼び出されるのを防ぐことができます。このようにするだけで、画面への表示速度が倍近く向上する場合があります。

高速化のためのその他のヒント

イベント トラップ

Visual Basic では、イベント ドリブン プログラミング モデルによってイベントを処理していますが、従来の Basic では、キー入力やシリアル ポートへの入力をイベントととらえ、割り込みによって処理を行っていました。この方法は、Visual Basic for MS-DOS でもサポートされています。このような処理を含むプログラムを (/V または /W オプションを指定して) コンパイルするとき、特に指定しなければ、プログラムの実行中に常にイベントが発生していないか監視されています。イベントのトラップが必要のない部分では、**EVENT OFF** ステートメントを使ってトラップをオフにしておくと、プログラム サイズが減るとともに、実行スピードも向上します。

行ラベル

行ラベルを付けると、その行での実行の最適化が制限されます。不必要な行ラベルを削除することによって、ある程度プログラムの実行スピードは上がります。

計算結果を格納した配列

計算量の多いプログラムの速度を上げるために、あらかじめ予想される入力について、関数や計算の結果を配列に保存しておく、ということがよく行われます。これは、配列の参照に必要な時間が実際の計算時間より短く、配列を保持するための余分なメモリが使用可能である場合にだけ有効です。例えば、タンジェントの計算が何度も必要な場合には、TAN (x) の結果を必要な種類だけあらかじめ計算しておき、配列に格納しておいた方が、毎回計算するよりははるかに高速です。

プロパティと変数

変数へのアクセスの方が、プロパティへのアクセスより高速です。プロパティを多用しているとき、可能なものを変数に置き換えることによって、速度が向上します。

C.3 その他のヒント

サイズやスピード以外にも、いくつかの一般的なテクニックがあります。

フォームのデザイン

フォームの大きさと位置をデザイン時に決定しようとすると、ツール ボックスやプロパティ バーがフォームに隠れ、作業のじゃまになるときがあります。このような場合、実行時 (フォームのロード時) に設定することもできます。

フォーム全体の色を制御するときは、**SCREEN** オブジェクトの ControlPanel プロパティを利用します。

デバッグ

イミディエイト ウィンドウで **MSGBOX** を実行すると、フォームの表示中に変数の内容を簡単に確認できます。

出力画面とプログラミング環境のユーザー インターフェイスの切り換えは、両者の表示行数が同じのときに、より高速に実行されます。そのようにしておけば、[F8] キー, [F10] キー などが高速になり、デバッグの効率が上がります。

プロジェクト中のフォーム モジュール

あるモジュールがフォームとしてロードされているかどうかを知るには、プロジェクト ウィンドウのファイル名の後に、フォーム名が表示されているかを確かめてください。フォーム名が表示されていなければ、フォーム モジュールではなくコード モジュールとして扱われています。

コードの記述

フォームに配置されているすべてのコントロールを知るには、プログラミング環境の [編集] メニューの [イベント プロシージャ] コマンドを選ぶと、オブジェクトの一覧が表示されます。各オブジェクトには、その種類や目的が分かるような名前を付けてください。また、プログラムの実行の間変化しないプロパティには、フォーム デザイナで値を設定して、設定のための余分なコードを省いてください。

実行中に変化するプロパティの内容は実行時に設定して、静的な文字列の量を減らしてください。フォーム デザイナでプロパティの値を設定しなければ、ロード時に静的な文字列領域が確保されません。

複数のモジュール間で共有するために **COMMON** ステートメントで宣言された変数は、変数名ではなく、宣言された順序とデータ型によって結び付けられます。これらのすべての変数が、プログラム中のすべてのモジュールで、同じ順序で宣言されていなければなりません。

個々のプロパティそのものをプロシージャに渡すことはできません。プロパティの値を渡すか、フォームまたはコントロール全体として渡します。

キーワード **PRINT** および **CLS** は、フォーム モジュール中で使われるとメソッドとして扱われ、コード モジュール中で使われるとステートメントとして扱われます。

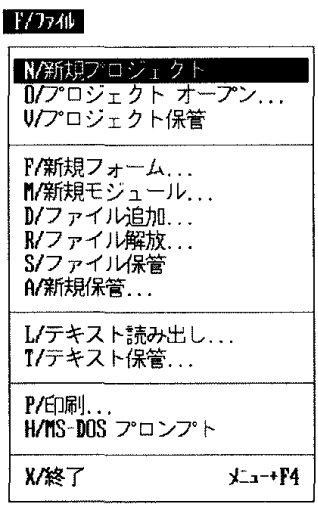
フォームまたはピクチャー ボックスに **PRINT** メソッドで何度か表示を行う場合、そのオブジェクトの AutoRedraw プロパティを真 (-1) に設定して、前に出力したテキストが再表示されるようにしてください。

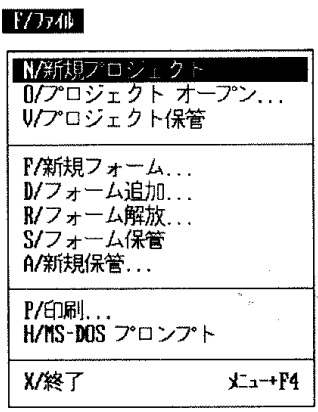
D Visual Basic のメニュー

Visual Basic のプログラミング環境とフォーム デザイナのメニューを説明します。

D.1 [ファイル] メニュー

[ファイル] メニューのコマンドは、フォーム モジュール，コード モジュール，プロジェクトを操作します。



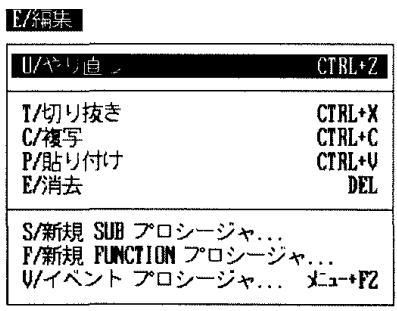


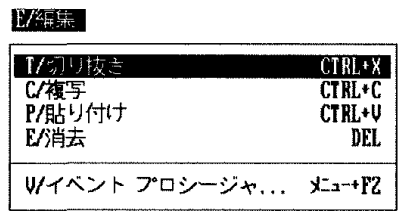
メニュー コマンド	機能
[新規プロジェクト], [プロジェクト オープン], [プロジェクト保管]	プロジェクトの新規作成，オープン，保管を行います。
[新規フォーム]	フォーム モジュール (.FRM) を新規作成します。
[新規モジュール]	コード モジュール (.BAS) を新規作成します。
[フォーム追加]	既存のフォーム モジュールをプロジェクトに追加します。
[ファイル追加], [ファイル解放]	コード モジュール，フォーム モジュール，インクルード ファイル，テキスト ファイルをプロジェクトに追加したり、解放します。
[ファイル保管]	現在編集中のファイルをディスクに保存します。
[新規保管]	新しい名前や書式に変更してから保存します。

[フォーム解放]	フォーム モジュールをプロジェクトから解放します。
[フォーム保管]	現在編集中のフォームをディスクに保存します。
[テキスト読み出し]	指定したテキスト ファイルから、アクティブなフォームまたはコード モジュールにコードを読み込みます。
[テキスト保管]	アクティブなフォームまたはコード モジュールのコードを、テキスト ファイルに保存します。
[印刷]	指定したテキスト、フォームまたはコード モジュール、アクティブなウィンドウの内容を、プリンタまたはファイルに出力します。
[MS-DOS プロンプト]	Visual Basic の実行を一時的に中断して、MS-DOS シェルを起動します。
[終了]	Visual Basic の実行を終了します。ただし、プログラミング環境からフォーム デザイナを起動しているときに [終了] コマンドを選ぶと、再びプログラミング環境がロードされます。

D.2 [編集] メニュー

[編集] メニューは、コードやコントロールの編集を行い、アプリケーションで使うプロシージャを作成します。

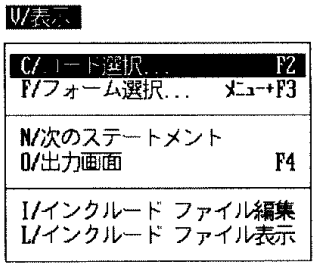


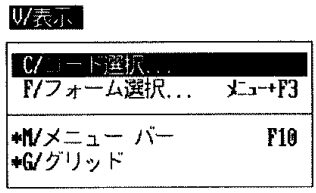


メニュー コマンド	機能
[やり直し]	最後に実行した編集操作をやり直します。
[切り抜き], [複写], [貼り付け]	編集対象の切り抜き，複写，貼り付けを行います。
[消去]	選択した項目を消去します。
[新規 SUB プロシージャ], [新規 FUNCTION プロシージャ]	SUB または FUNCTION プロシージャを新規に作成します。
[イベント プロシージャ]	イベント プロシージャを作成または表示します。

D.3 [表示] メニュー

[表示] メニューは、コード モジュール，プロシージャ，フォームの表示と選択を行います。

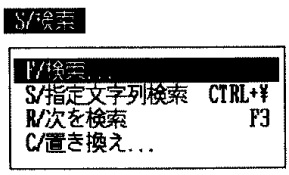




メニュー コマンド	機能
[コード選択]	指定したプロシージャを表示します。
[フォーム選択]	指定したフォームを表示します。
[メニューバー]	フォーム デザイナのメニュー バーの表示/非表示を切り換えます。
[グリッド]	フォーム デザイナのグリッド (格子) の表示/非表示を切り換えます。
[次のステートメント]	次に実行するステートメントを表示します。
[出力画面]	アプリケーションの出力を画面全体に表示します。
[インクルード ファイル編集]	カーソルが **$INCLUDE** メタコマンドの行にあるときに、インクルード ファイルを読み込んで編集できる状態にします。
[インクルード ファイル表示]	アクティブ ウィンドウ中におけるインクルード ファイルの表示/非表示を切り換えます。

D.4 [検索] メニュー

[検索] メニューは、コード ウィンドウにおける検索と置き換えを行います。



メニュー コマンド	機能
[検索]	指定したテキストの検索と表示を行います。
[指定文字列検索]	選択 (反転表示) されているテキストの検索と表示を行います。

[次を検索]	指定または選択したテキストが次に登場する箇所を検索します。
[置き換え]	指定したテキストを検索して、別のテキストと置き換えます。

D.5 [実行] メニュー

[実行] メニューは、プログラムの実行と実行ファイルの作成を行います。

R/実行

```
S/スタート          SHIFT+F5
R/リスタート
N/継続                  F5
C/COMMAND$ 編集...

X/EXE ファイル作成...
L/ライブラリ作成...

F/実行開始ファイル設定...
```

メニュー コマンド	機能
[スタート]	カレントのアプリケーションを実行します。
[リスタート]	カレントのアプリケーションの実行を再開始します。
[継続]	停止していたアプリケーションの実行を再開します。
[COMMAND$ 編集]	コマンド ラインの引数を指定します。
[EXE ファイル作成]	ロードされているアプリケーションの実行ファイル (.EXE) を作成します。
[ライブラリ作成]	そのときロードされているすべてのモジュールから Quick ライブラリ (.QLB) とオブジェクト モジュール ライブラリ (.LIB) を作成します。
[実行開始ファイル設定]	実行を開始するモジュールを設定します。

D.6 [ﾃﾞﾊﾞｯｸﾞ] メニュー

[ﾃﾞﾊﾞｯｸﾞ] メニューは、コード内のエラーを探すための機能を提供します。

D/ﾃﾞﾊﾞｯｸﾞ

A/ウォッチ追加...	
I/インスタント ウォッチ...	SHIFT+F9
W/ウォッチ ポイント...	
D/ウォッチ解除...	
L/全ウォッチ解除	
T/トレース ON/OFF	
H/ヒストリ ON/OFF	
B/ブレークポイント ON/OFF	F9
C/全ブレークポイント解除	
E/エラー時中断 ON/OFF	
S/次のステートメント設定	

メニュー コマンド	機能
[ウォッチ追加]	プログラムのステップ実行中に参照できるように、デバッグ ウィンドウに式の値を表示します。
[インスタント ウォッチ]	指定した式の現在の値をダイアログ ボックスに表示します。
[ウォッチ ポイント]	式が True (0 以外の値) になったときにアプリケーションの実行を停止します。
[ウォッチ解除]	デバッグ ウィンドウから 1 つの式を削除します。
[全ウォッチ解除]	デバッグ ウィンドウのすべての式を削除します。
[トレース ON/OFF]	実行中のステートメントを強調表示しながら実行します。
[ヒストリ ON/OFF]	実行の経過を記録します。 [SHIFT]+[F8] キーまたは [SHIFT]+[F10] キーで、ヒストリをさかのぼることができます。
[ブレーク ポイント ON/OFF]	ブレークポイントの設定と解除を行います。プログラムの実行は、ブレークポイントを設定したステートメントで停止します。
[全ブレーク ポイント解除]	すべてのブレークポイントを解除します。
[エラー時中断 ON/OFF]	エラー処理ルーチンの最初のステートメントで、プログラムの実行が停止するようにします。
[次のステートメント設定]	次に実行するステートメントを設定します。

D.7 [ツール] メニュー

[ツール] メニューは、フォーム デザイナでフォーム上に描くコントロールを選ぶために使います。キーボードで操作する場合に使います。マウスでツール ウィンドウを操作するのと同じ機能です。

T/ツール

L/ラベル
C/コマンド ボタン
O/オプション ボタン
K/チェック ボックス
X/テキスト ボックス
S/リスト ボックス
M/コンボ ボックス
P/ピクチャー ボックス
A/フレーム
T/タイマー
H/水平スクロール バー
V/垂直スクロール バー
D/ディレクトリ リスト
R/ドライブ リスト
F/ファイル リスト

D.8 [補助] メニュー

[補助] メニューは、プログラミング環境とフォーム デザイナの画面と動作を設定します。

O/補助

D/画面設定...
P/パス設定...
M/マウス設定...
V/保管条件...
*S/構文チェック ON/OFF

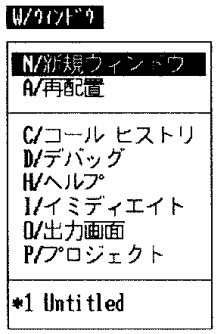
O/補助

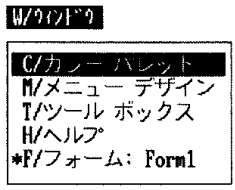
D/画面設定...
P/パス設定...
V/保管条件...

メニュー コマンド	機能
[画面設定]	プログラミング環境とフォーム デザイナの画面の色などを設定します。プログラミング環境では、タブ ストップの設定も行います。
[パス設定]	実行可能ファイル，インクルード ファイル，ライブラリ，ヘルプ ファイルのパスを設定します。
[マウス設定]	マウスの右ボタンの操作を設定します。
[保管条件]	バックアップと保管のオプションを設定します。
[構文チェック ON/OFF]	構文チェック機能のオン /オフを切り換えます。

D.9　[ｳｨﾝﾄﾞｳ] メニュー

[ｳｨﾝﾄﾞｳ] メニューでは、表示するウィンドウを選びます。





メニュー コマンド	機能
[新規ウィンドウ]	コード ウィンドウを新規作成します。
[再配置]	ウィンドウをタイル表示に再配置します。
[コール ヒストリ], [デバッグ], [ヘルプ], [イミディエイト], [出力画面], [プロジェクト]	プログラミング環境で、指定したウィンドウをオープンします。
[カラー パレット], [メニュー デザイン], [ツール ボックス], [ヘルプ], [フォーム]	フォーム デザイナで、指定したウィンドウをオープンします。

D.10 [ﾍﾙﾌﾟ] メニュー

[ﾍﾙﾌﾟ] メニュー は、Visual Basic のオンライン ヘルプを表示します。

H/ﾍﾙﾌﾟ

I/索引 C/目次
K/キーボード
T/トピック:　F1 H/ヘルプの使い方　SHIFT+F1
A/Visual Basic について...

メニュー コマンド	機能
[索引]	Visual Basic ヘルプの索引を表示します。
[目次]	Visual Basic ヘルプの目次を表示します。
[キーボード]	キーボード操作の一覧を表示します。
[トピック]	カーソル位置のキーワードに関するヘルプ トピックを表示します。
[ヘルプの使い方]	Visual Basic ヘルプの使い方を表示します。
[Visual Basic について]	バージョン番号，著作権，所有者の情報を表示します。

索引

数字・記号

, (カンマ)
　出力行中の 147
... (省略記号) 159
; (セミコロン)
　出力行中の 147
? (疑問符) 207, 235
! (階乗記号) 351
| (縦棒)
　構文中の xxv
() (カッコ)
　SUB プロシージャの呼び出し 326
　値による引き渡し 333
　論理式中の 305
[] (角カッコ)
　構文中の xxiv
{} (大カッコ)
　構文中の xxv
= (演算子) 304
< (演算子) 304
<= (演算子) 304
<> (演算子) 304
> (演算子) 304
>= (演算子) 304
& (アンパサンド)
　アクセス キー 53, 163
⇒ (行連結文字)
　プログラム例中の xxv

A

About プロシージャ
　構文 113
ActiveControl プロパティ 57
ADDITEM メソッド 76
AND 演算子 304
　ビットフィールドの演算 278
ANY キーワード 331, 336, 340, 385
AS キーワード
　AS *type* 節 328, 329, 331, 332, 350, 385
　ファイル番号の宣言 244
ASCII フォーム 401
AUTOEXEC.BAT ファイル 3
AutoRedraw プロパティ 280
AutoSize プロパティ 96

B

BackColor プロパティ 41, 297, 405
.BAS モジュール 17, 29, 126
　→「コード モジュール」も参照
　.MAK ファイル中 124
　Quick ライブラリへの組み込み 390, 394
　バイナリ形式とテキスト形式 128
Basic コード
　→「コード モジュール」も参照
　イベント プロシージャ 30, 46
　ジェネラル プロシージャ 30, 46
　追加 132
　入力 23, 45
　モジュール レベル 29, 127
Basic プログラム
　イベントドリブン 32
Basic モジュール
　→「コード モジュール」を参照
BC.EXE 3
　Quick ライブラリの作成 391, 394
　インストール 3
　エラー処理オプション 202
BEGIN
　ASCII フォーム 402
.BI ファイル 127
　→「インクルード ファイル」も参照
　保管 129
B_OnExit ルーチン 396
BorderStyle プロパティ 67
Button 引数 276
BYVAL キーワード 334

C

CALL Absolute ルーチン 129, 388
CALL Interrupt ルーチン 129, 388
CALL InterruptX ルーチン 129, 388
CALL ステートメント 46, 326
Cancel プロパティ 117
Caption プロパティ 41
　ASCII フォーム 405
　アクセス キー 53, 164, 209
　コマンド ボタン 62
　設定 22
CASE キーワード 311
CASE ELSE キーワード 311
CHAIN ステートメント 249
Change イベント 77, 86, 207, 212

ChangeText プロシージャ 113
CHDIR ステートメント 206
CHDRIVE ステートメント 204
CLEAR メソッド 146
Click イベント 31, 77, 83, 86
　発生の条件 32
　メニュー コントロール 158
　リスト ボックスの例 83
CLIPBOARD オブジェクト 36, 141
　クリア 146
　メソッド 142, 146
CLOSE ステートメント 242, 249
CLS ステートメント 146, 226
CLS メソッド 146, 226
　PRINT メソッドの出力の消去 150
CMNDLG.BAS モジュール
　[ファイル オープン] の例..... 109, 113, 180, 214
CMNDLG.BI ファイル 214
CMNDLGF.FRM モジュール
　[ファイル オープン] の例..... 109, 180, 214
ColorPalette プロシージャ 113
COM デバイス 266
COMMAND$ 関数 381
[COMMAND$ 編集] コマンド ([実行] メニュー)
　..... 381, 434
COMMON ステートメント
　SHARED 属性..... 342, 347
　インクルード ファイル 129, 339
　名前付きブロック 347
　メモリ消費 420
CONS デバイス 266
CONST ステートメント
　インクルード ファイル 129, 339
　メモリ消費 420
CONSTANT.BI ファイル
　..... 127, 129, 284, 286, 299
CONTROL キーワード 332
ControlPanel サンプル アプリケーション
　使い方 301
ControlPanel プロパティ 60, 166, 297
　記号定数 298
CRSLN ステートメント 227, 239
CtlName プロパティ 41, 60
CURRENCY データ型 425
CurrentX プロパティ 149
CurrentY プロパティ 149

D

DblClick イベント 32, 76, 83, 209
DECLARE ステートメント
　ANY キーワード 331, 336, 340, 385
　BYVAL キーワード..... 334
　SUB プロシージャ..... 327
　自動生成 337
　パラメータ リスト..... 327
　引数のチェック 335
　複数のモジュールによるアプリケーション
　..... 323
　別のエディタでの処理 337
　メモリ消費 420
　ユーザー定義型 331, 336, 339
DEF FN プロシージャ 425
DEF*type* 系ステートメント 367
DGROUP 416, 418, 420
DIM SHARED ステートメント 342, 344
DIR$ 関数 189
Disabled プロパティ 118
DO...LOOP ステートメント 317
DOEVENTS 関数 383
DRAG メソッド 289
DragDrop イベント 289
DragMode プロパティ 289
DragOver イベント 289
Drive プロパティ 204
DropDown イベント 86

E

/Ea コマンドライン オプション 397
ELSEIF キーワード 308
EMS メモリ 395, 397, 411, 412, 415
　使い方 395
　プログラミング環境 411, 415
Enabled プロパティ
　設定 118
　タイマー コントロール..... 103
　フォーカス 56
　メニュー コントロール..... 164
END
　ASCII フォーム 403
END SUB ステートメント..... 24, 48
END ステートメント
　エラー処理ルーチン 191
　使い方 24, 51, 173, 191
ENDDOC メソッド..... 153
EOF 関数 171, 243, 267, 269
EQV 演算子..... 304
ERR 関数 184, 197, 199, 201
ERROR ステートメント 202
ERROR$ 関数 184
ERROR ERR ステートメント 191, 195, 200
EVENT OFF..... 428
EVENT ON..... 428
.EXE ファイル
　→「実行可能ファイル」を参照
[EXE ファイル作成] コマンド ([実行] メニュー)
　..... 25, 434
[EXE ファイル作成] ダイアログ ボックス..... 25
EXIT DO ステートメント 319

EXIT FOR ステートメント 316
EXIT FUNCTION ステートメント
..... 186, 193, 325
EXIT SUB ステートメント 186, 193, 325

F

Far データ ヒープ
記述子のメモリ消費 420
実行可能ファイル 417
プログラミング環境 415
FD.EXE
→「フォーム デザイナ」を参照
FileName プロパティ 220
FileOpen プロシージャ 113, 216
FilePrint プロシージャ 113
FileSave プロシージャ 113
FindText プロシージャ 113
FOR...NEXT ステートメント 313
FORMAT$ 関数..... 45, 65, 66, 151, 230
FormName プロパティ 41
FormType プロパティ 176, 405
FORM キーワード 332
$FORM メタコマンド 119, 178, 349
FRE 関数 354, 421
FREEFILE 関数..... 243, 245, 247
.FRM モジュール
→「フォーム モジュール」を参照
FUNCTION プロシージャ 45
DECLARE ステートメント 338
インクルード ファイル 339
再帰 352
呼び出し 326, 426
ローカル変数の使用 423

G

GET ステートメント
ファイル I/O..... 260
GETTEXT メソッド 142
GotFocus イベント 56

H

Height プロパティ 96
Hello world アプリケーション 17
HIDE メソッド 118

I

IF...THEN...ELSE ステートメント
構文 307
IF...TYPEOF ステートメント 57
IMP 演算子 304
INCLUDE 環境変数 3
$INCLUDE メタコマンド
..... 119, 129, 188, 299, 340, 342
Index プロパティ 98, 103, 168
.INI ファイル..... 301
INKEY$ 関数..... 226, 236
INPUT ステートメント 227, 234
INPUT # ステートメント 243, 269
INPUT$ 関数 227, 236, 269
INPUTBOX$ 関数 108, 109, 173
Interval プロパティ 102

K

KeyAscii 引数 285
KEYBD デバイス 266
KeyCode 引数..... 286
KeyDown イベント 285, 382
KeyPress イベント 31, 137, 285
KeyUp イベント 285
KILL ステートメント 257
KLEN 関数..... 140, 143

L

/L コマンドライン オプション..... 387
LargeChange プロパティ 91
LBOUND 関数..... 330
Left プロパティ..... 93
LEN キーワード 248
LEN 関数 140
LIB 環境変数..... 3
LIB.EXE
インストール 3
ライブラリの作成 391
.LIB ファイル 127, 130
Quick ライブラリの作成 392
作成 388, 391
LINE INPUT ステートメント 172, 242, 269
LINE INPUT # ステートメント 233, 242
LineDraw アプリケーション 280
LINK.EXE
Quick ライブラリの作成 391
インストール 3
List プロパティ 77, 80
ListCount プロパティ 77, 79, 81, 206
ListIndex プロパティ 77, 80, 81, 205
LOAD ステートメント 100, 102, 118, 170

LOC 関数 267, 269
LOCAL キーワード 185
LOCATE ステートメント 226, 231, 233, 239
LOF 関数 243, 267, 269
LostFocus イベント 56
LPT デバイス 266

M

.MAK ファイル 121, 127
　インクルード ファイル 340
MASM
　Quick ライブラリの作成 395
Max プロパティ 91
MDI アプリケーション 173
Microsoft C/C++
　Quick ライブラリの作成 395
Microsoft Macro Assembler
　Quick ライブラリの作成 395
Microsoft QuickBasic
　vs. Visual Basic 32
Microsoft Visual Basic のコンパイラ 3
　→「BC.EXE」も参照
Min プロパティ 91
MOUSE.BAS ファイル 294
MouseDown イベント 271, 277, 281
MouseMove イベント 31, 271, 276, 282, 284
MousePointer プロパティ 274
MouseUp イベント 271, 277, 281
MOVE メソッド 93, 292
[MS-DOS プロンプト] コマンド ([ﾌｧｲﾙ] メニュー) 431
MSGBOX 関数 26, 108, 110
MSGBOX ステートメント 108, 110
MultiLine プロパティ 135

N

NAME ステートメント 199
Near データ ヒープ
　→「DGROUP」を参照
NEWPAGE メソッド 153
NEW-VARS.BAT ファイル 3
NOT 演算子 304
NOW 関数 152

O

.OBJ ファイル 393
ON ERROR GOTO ステートメント 184, 191, 200, 202
ON ERROR RESUME NEXT ステートメント 197, 199
ON LOCAL ERROR GOTO ステートメント 191, 199, 200
ON LOCAL ERROR RESUME NEXT ステートメント 197, 199, 201
OPEN ステートメント
　LEN= 引数 242, 427
　エラー トラップ 195
　構文 244
　シーケンシャル アクセス 249
　デバイスのアクセス 266
　バイナリ アクセス 261
　ファイル アクセス 243
　ランダム アクセス 258
OPTION EXPLICIT ステートメント 368
OR 演算子 304

P

PACKING.TXT ファイル 4
PATH 環境変数 3
Pattern プロパティ 207, 221
POS ステートメント 227, 239
PRINT ステートメント
　vs. PRINT メソッド 146, 226
　イミディエイト ウィンドウ 380
　画面への表示 228
　機能 226
　速度優先の最適化 427
PRINT メソッド
　Form_Load プロシージャでの使用 51
　vs. PRINT ステートメント 146, 226
　イミディエイト ウィンドウ 380
　機能 226
　構文 52, 146
　ピクチャー ボックス 88
　プリンタへの出力 253
PRINT USING ステートメント 230
PRINTER オブジェクト 36
　指定 152
PRINTFORM メソッド 153
PrintTarget プロパティ 152
PUT ステートメント
　ファイル I/O 260

Q

.QLB ファイル 127
→「Quick ライブラリ」も参照
ファイル名拡張子 393
QLBVIEW サンプル アプリケーション 363
Quick ライブラリ 127, 130, 387
VBDOS.QLB 130, 388
作成 390
作成時の注意 396
必要なファイル 388, 391
プロシージャの宣言 342
モジュールの追加 394
[ライブラリ作成] ダイアログ ボックス..... 393
ロード 387, 389

R

RAM ディスク 414
→「メモリ」も参照
REDIM SHARED ステートメント 342
REMOVEITEM メソッド..... 77
RESUME ステートメント
..... 184, 186, 193, 199, 202
RESUME NEXT ステートメント
..... 186, 191, 199, 202

S

/S コマンドライン オプション..... 411
ScaleHeight プロパティ 151
ScaleWidth プロパティ..... 151
SCREEN オブジェクト 36, 233
ActiveControl プロパティ 57
ControlPanel プロパティ 60, 166, 297
SCRN デバイス..... 266
ScrollBars プロパティ 90, 135
SEEK 関数
シーケンシャル アクセス ファイル
..... 263, 266
SEEK サンプル アプリケーション 208
SEEK ステートメント
シーケンシャル アクセス ファイル
..... 263, 266
SELECT CASE...END SELECT ステートメント
..... 309
SelLength プロパティ..... 142
SelStart プロパティ 142
SelText プロパティ..... 142
Separator プロパティ 163
SETFOCUS メソッド..... 50, 56
SETTEXT メソッド 142
SHARED キーワード
変数の共有 344
SHARED ステートメント 343
Shift 引数
キーボード イベント..... 287
マウス イベント..... 276
SHOW メソッド 108, 118, 178, 302
SmallChange プロパティ 91
Sorted プロパティ 80, 81
Source 引数 289
SPC ステートメント 227
STACK 関数 354
STACK ステートメント 354
State 引数..... 291
STATIC キーワード 325, 350, 352, 427
STATIC ステートメント 350
[STOP] キー..... 369
STR$ 関数 65
Style プロパティ 75, 86
SUB ステートメント 25, 48
SUB プロシージャ 45
DECLARE ステートメント..... 338
呼び出し 326, 426
ローカル変数 423

T

TAB 関数..... 149, 226, 230
[TAB] キー
→「タブ オーダー」も参照
テキスト ボックスでの使い方..... 257
TabIndex プロパティ 54, 117, 403
TabStop プロパティ 55, 117, 257
Text プロパティ 41, 65, 76, 80
TEXTHEIGHT メソッド 150
TEXTWIDTH メソッド 150
TMP 環境変数..... 414
Top プロパティ..... 93
TSR プログラムとメモリ 416, 418
TYPE...END TYPE ステートメント
ANY キーワード 258, 330, 336

U

UBOUND 関数 330
UCASE$ 関数 74, 138
UNLOAD ステートメント..... 100, 102, 118, 170
UNTIL キーワード 317

V

VAL 関数 44, 66
Value プロパティ 91
VBDOS.BI ファイル 129
VBDOS.EXE ファイル 391
→「プログラミング環境」も参照
VBDOS.LIB ライブラリ 131, 390
VBDOS.QLB Quick ライブラリ 131, 388
VBDOSQLB.LIB ライブラリ 388, 391
VBDRT10.EXE ランタイム モジュール 130
VBDRT10.LIB ランタイム ライブラリ 130
VIEW PRINT ステートメント 227, 231
Visible プロパティ 56, 97, 118, 165
Visual Basic
vs. Microsoft QuickBasic 32
インストール 3
オンライン ヘルプ 12
ランタイム モジュール 3, 130
[Visual Basic について] コマンド ([ﾍﾙﾌﾟ] メニュー) 438

W

WHILE キーワード 317
WHILE...WEND ステートメント
vs. DO...LOOP ステートメント 319
Width プロパティ 96
WRITE # ステートメント 243

X

XMS メモリ
プログラミング環境 412, 415
XOR 演算子 304

ア

アクセス キー
→「ショートカット キー」を参照
アクティブ ウィンドウ 7
影のオン/オフ 297
選択 9
アセンブリ言語ルーチン 129, 395
アプリケーション
vs. プロジェクト 121
vs. モジュール 126
イベントドリブン 29
外観の変更 297
コード モジュール 17, 29, 126
実行の停止 368
指定部分の実行 374
設計の方法 365
デバッグ 357
フォームの作成 36
フォーム モジュール 17, 29, 127
アプリケーション (サンプル)
ControlPanel 301
LineDraw 280
Quick ライブラリ ビューア 390
Try Menus 159
アラーム クロック 105
端末エミュレータ 269
テキスト エディタ 169, 176, 250
デジタル クロック 104
ファイル検索 208
アプリケーションのデバッグ
OPTION EXPLICIT 368
イベントに関する注意点 382
イミディエイト ウィンドウ 379
ウォッチ式 373
エラーの種類 357
コマンドの一覧 360
ショートカット キー 360
ステップ実行 375
プログラムの実行の停止 368
変数やプロシージャの表示 364
アンパサンド (&) 53, 163

イ

移動
ウィンドウ 8
オブジェクト 39, 45
コントロール 21
イベント 31
[イベント プロシージャ] ダイアログ ボックス 24, 47
キーボード 285
コンボ ボックス 86
ドラッグ & ドロップ 293
ファイル システム 217
マウス 271
リスト ボックス 83
連鎖反応 383
イベントドリブン アプリケーション 31
vs. 従来の Basic アプリケーション 32
インターフェイスの設計 35
概要 31
作成 29
終了 51
デバッグに関する注意点 382
特有のエラー 382

イベント プロシージャ 23
Basic コード 30, 45
機能 30
構造化プログラミング 321
構文 48
作成 47
名前の付け方 48
パラメータ リスト 49
[イベント プロシージャ] コマンド([編集メニュー])
..... 6, 12, 24, 47
[イベント プロシージャ] ダイアログ ボックス
..... 24, 47
イミディエイト ウィンドウ 7, 362, 437
値の代入 379
キー操作 380
コードの実行 377
スコープ 378
プロシージャの呼び出し 379
色
ASCII フォーム 405
ControlPanel プロパティによる変更 297
インクルード ファイル 126, 129
CMNDLG.BI 216
CONSTANT.BI 129
DECLARE ステートメント 337
DECLARE ステートメントの自動生成
..... 339
VBDOS.BI 129
宣言 339
内容の表示 339
プロシージャ宣言 339
保管 128
モジュール 340
利点 339
[インクルード ファイル表示] コマンド
([表示] メニュー) 363, 433
[インクルード ファイル編集] コマンド
([表示] メニュー) 363, 433
印刷
ソース コード 133
フォーム 133
[印刷] コマンド ([ﾌｧｲﾙ] メニュー) 133, 431
[インスタント ウォッチ] コマンド
([ﾃﾞﾊﾞｯｸﾞ] メニュー) 361, 373, 435

ウ

ウィンドウ
アクティブ 9
種類 7
操作 8
[ｳｨﾝﾄﾞｳ] メニュー 362, 437
[イミディエイト] コマンド/ウィンドウ
..... 437
ウィンドウの種類 7
[カラー パレット] コマンド 437
[コール ヒストリ] コマンド 362
[コール ヒストリ] コマンド/ウィンドウ
..... 437
[再配置] コマンド 437
[出力画面] コマンド 362
[出力画面] コマンド/ウィンドウ 437
[新規ウィンドウ] コマンド 437
[ツール ボックス] コマンド 437
[デバッグ] コマンド 362
[デバッグ] コマンド/ウィンドウ 437
[プロジェクト] コマンド 362
[プロジェクト] コマンド/ウィンドウ 437
[ヘルプ] コマンド 362, 437
[ヘルプ] コマンド/ウィンドウ 437
[メニュー デザイン] コマンド 437
[ウォッチ解除] コマンド ([ﾃﾞﾊﾞｯｸﾞ] メニュー)
..... 361, 371, 374, 435
[ウォッチ追加] コマンド ([ﾃﾞﾊﾞｯｸﾞ] メニュー)
..... 361, 373, 435
ウォッチポイント 371
[ウォッチ ポイント] コマンド ([ﾃﾞﾊﾞｯｸﾞ] メニュー)
..... 361, 371, 435

エ

エラー
構文エラー 358
ランタイム エラー 183, 358
論理エラー 359
[エラー時中断 ON/OFF] コマンド
([ﾃﾞﾊﾞｯｸﾞ] メニュー) 361, 435
エラー処理
あとで実行 197, 200
エラーの種類 189, 357
コード中の位置 189
コンパイル 202
終了 189
汎用的なプロシージャ 195
複雑なプログラム 194
プログラムの再開位置 194
プログラムの制御への応用 201
プロシージャ レベル 188, 191
モジュール レベル 191, 200
無効化 200
ユーザー定義のエラー コード 202
呼び出しパス 192

オ

[置き替え] コマンド ([検索] メニュー)..... 433
オーバーヘッド 419
オブジェクト
　移動 39, 45
　イベントに対する応答 31
　[イベント プロシージャ] ダイアログ ボックス 24
　イベント プロシージャの作成 47
　サイズの変更 39, 45
　参照 35
　選択 38
　特殊オブジェクト 35
　ドラッグ 288, 291
　ドロップ 289
　フォーカス 54
　プロパティの設定 21, 40
オブジェクト モジュール ファイル (.OBJ) 393
オブジェクト モジュール ライブラリ ファイル (.LIB)..... 126
　VBDOS.LIB..... 131, 390
　作成 391, 397
オプション ボタン..... 59
　TabStop プロパティ 55
　グループ化 55, 69
　フォームへの配置 20
親コントロール 61
オンライン ヘルプ..... 12, 16
　キーワードと [F1] キー 364
　表示 13
　複写と貼り付け 15
　変数の定義とスコープ 364
　メニューから呼び出し 364

カ

階乗関数の例 351
カーソル
　位置の指定 238
　位置の取得 239
　形状の変更 239
型宣言 329
[画面設定] コマンド ([補助] メニュー)..... 436
カラー パレット (フォーム デザイナ) 12
[カラー パレット] コマンド ([ｳｨﾝﾄﾞｳ] メニュー) 437
環境変数 3
関係演算子 304
カンマ (,)
　出力行中 147

キ

記号定数
　オンライン ヘルプ..... 365
　プロシージャへの引き渡し 329
記述子 420
基数表示の例 71
起動
　概要 3
　コマンドライン オプション..... 4
　実行開始ファイル 133
　プロジェクトの指定 123
キーボード イベント 284
キーボード インターフェイス
　アクセス キーの割り当て..... 52
　タブ オーダー..... 54
　入力 233
　マウス操作 17
　メニュー コントロール..... 163
[キーボード] コマンド ([ﾍﾙﾌﾟ] メニュー)..... 438
行ラベル
　エラー処理ルーチン 188
　削除 428
[切り抜き] コマンド ([編集] メニュー)..... 432

ク

[グリッド] コマンド ([表示] メニュー)..... 433
クリップボード 141
グローバル変数 324

ケ

[継続] コマンド ([実行] メニュー)..... 360, 370, 434
[検索] メニュー
　[置き替え] コマンド 433
　[検索] コマンド 433
　[指定文字列検索] コマンド 433
　[次を検索] コマンド 433

コ

構造化プログラミング 321
構文
　マニュアルの表記規則 xxiv
構文エラー 358
　ヘルプの表示 364
[構文チェック ON/OFF] コマンド ([補助] メニュー) 364, 436
子コントロール 61
コード
　入力 23, 46
　プログラミング環境外でのコンパイル 337
　マージ/置き換え..... 132

コード ウィンドウ 5, 7
[コード選択] ダイアログ ボックス 50
[コード選択] コマンド ([表示] メニュー) 12, 390, 433
コード モジュール 17, 29, 126
FUNCTION プロシージャ 127
PRINT ステートメント 146
Quick ライブラリへの組み込み 392
SUB プロシージャ 127
イベント プロシージャ 30, 46
ジェネラル プロシージャ 30, 46
実行開始ファイル 133
バイナリ形式とテキスト形式 128
モジュール レベル コード 29, 126
コード例
DECLARE ステートメント 336, 337
DO...LOOP ステートメント 318, 319
Hello world 23
IF...THEN...ELSE ステートメント 308
LineDraw 281
SELECT CASE...END SELECT ステートメント 310
SUB プロシージャ 326, 338
アラーム クロック 105
インクルード ファイル 341
エラー 358
温度換算 67
階乗関数 352
基数表示 71
キーボード イベント 287
クリップボード 144
コモン ダイアログ ツールキット 218
コントロール配列 101
再帰プロシージャ 353
端末エミュレータ 270
チェック ボックス 74
テキスト エディタ 169, 253
デジタル クロック 104
電卓 68, 181
投資計算 65
ドラッグ & ドロップ イベント 294
バイナリ ファイル アクセス 263
パスワード 139
ピクチャー ボックス 89
表データの出力 148
ファイル検索 209
マルチ ドキュメント インターフェイス (MDI) 177
メニュー 161, 171
メモ帳 181
リスト ボックス 78, 85
子フォーム
MDI アプリケーション 173
テキスト エディタ サンプル アプリケーション 175, 176
デスクトップ サンプル アプリケーション 179
コマンド ボタン 17, 59
フォームへの配置 19
プロパティの一覧 40
プロパティの設定 62
目的 62
コマンドライン
/E オプション 412
/Ea オプション 397
/L オプション 4, 387, 388
Quick ライブラリのロード 4, 387, 389
/S オプション 411
/X オプション 412
実行可能ファイルの作成 125
コマンドライン引数
使い方 381
テスト 381
コメント文字 (') 129
コモン ダイアログ ツールキット 112, 180, 214
[ファイル オープン] の例 216
コール ヒストリ ウィンドウ 7, 189
[コール ヒストリ] コマンド/ウィンドウ ([ｳｨﾝﾄﾞｳ] メニュー) 362, 437
コントロール 17, 35, 59
ASCII フォーム 403
アクセス キーの割り当て 52, 163
アクティブ 56
一覧 59
移動 21
エラー 383
親子関係 61
オンライン ヘルプ 60
グループ 87, 98
サイズの変更 20
削除 40
実行時の移動 92
実行時のサイズの変更 96
実行時の操作 37
実行時の表示 97
スクロール バー 90
タブ オーダー 54
名前の付け方 60
表示/非表示の切り換え 97
ファイル システム 203
フォーカス 54
フォームへの配置 19, 37
プロシージャへの引き渡し 332
プロパティの設定 21, 41
メニュー 155
メモリ消費 421
コントロール配列
実行時の作成 100, 168
設計時の作成 99
メニューの例 168
コントロール ブロック 403
コントロール ボックス 8

コンボ ボックス 59
Style プロパティの設定 76, 86
イベント 80, 86
項目の追加と削除 79
内容の操作 80
プロパティ 80, 86
メソッド 80

サ

再帰プロシージャ 351
最小化 8
サイズ
最適化 423
サイズの変更
ウィンドウ 8
オブジェクト 39, 45
コントロール 20
サイズ ハンドル 20, 39
最大化 8, 39
[再配置] コマンド ([ｳｨﾝﾄﾞｳ] メニュー) 437
[索引] コマンド ([ﾍﾙﾌﾟ] メニュー) 438
座標
ASCII フォーム 405
テキスト画面 227
マウスの座標 293
サンプル アプリケーション
→「アプリケーション (サンプル)」を参照

シ

ジェネラル プロシージャ 30, 46
構造化プログラミング 322
初期化 51
式
ウォッチ式 373
プロシージャへの引き渡し 329, 332
論理式 304
シーケンシャル アクセス ファイル 243
SEEK の使い方 263
速度優先の最適化 427
テキスト ファイル 250
時刻の書式 152
[COMMAND$ 編集] コマンド 381, 434
[EXE ファイル作成] コマンド 25, 434
[継続] コマンド 360, 370, 434
[実行開始ファイル設定] コマンド 434
[スタート] コマンド 25, 434
[ライブラリ作成] コマンド 390, 393, 434
[リスタート] コマンド 360, 434
[指定文字列検索] コマンド ([検索] メニュー)
..... 433
実行可能ファイル
コマンドラインから作成 125
コンパクト 397
作成 121
種類 130
プログラミング環境で作成 23, 121
メモリ 411
ランタイム ライブラリとのリンク 130
実行時
コントロールの操作 92
メニュー コントロールの追加 169
[終了] コマンド ([ﾌｧｲﾙ] メニュー) 12, 431
出力
出力位置の指定 149
速度優先の最適化 427
デスクトップ 228
ピクチャー ボックス 146
表示領域 148
フォーム 146
プリンタ 152
出力画面ウィンドウ 7
[出力画面] コマンド ([表示] メニュー) 363, 433
[出力画面] コマンド/ウィンドウ
([ｳｨﾝﾄﾞｳ] メニュー) 362, 437
出力のリダイレクト 227
[消去] コマンド ([編集] メニュー) 432
条件判断 307
省略記号 (...) 159
書式
時刻 152
数値 151
日付 152
ショートカット キー
→「アクセス キー」も参照
ASCII フォーム 405
メニュー コントロール 163
シリアル ポート 269
[新規 FUNCTION プロシージャ] コマンド
([編集] メニュー) 432
[新規 SUB プロシージャ] コマンド
([編集] メニュー) 432
[新規ウィンドウ] コマンド ([ｳｨﾝﾄﾞｳ] メニュー)
..... 437
<新規ウィンドウ編集> ボタン 7, 25, 48
[新規フォーム] コマンド ([ﾌｧｲﾙ] メニュー)
..... 6, 10, 18, 36, 431
[新規プロジェクト] コマンド ([ﾌｧｲﾙ] メニュー)
..... 18, 37, 122, 431
[新規保管] コマンド ([ﾌｧｲﾙ] メニュー) 431
[新規モジュール] コマンド ([ﾌｧｲﾙ] メニュー)
..... 431
シングル ステップ実行 375

ス

垂直スクロール バー コントロール 59
水平スクロール バー コントロール 59
数値
　PRINT ステートメントによる表示 228
　PRINT USING ステートメントによる表示 230
　書式 151, 230
数値定数
　プロシージャへの引き渡し 329
スクロール バー
　コントロール 90
　テキスト ボックス 135
スクロール ボックス 91
スタック 354
[スタート] コマンド ([実行] メニュー)..... 25, 434
スタンドアロン ライブラリ
　→「オブジェクト モジュール ライブラリ ファイル (.LIB)」を参照
ステータス バー 5
ステートメント
　ステップ実行 375
　非実行可能ステートメントのメモリ消費 420
スペース
　出力位置の調整 230

セ

制御構造 303, 425
　DEF FN ブロック 426
　FUNCTION プロシージャ 321, 420
　SUB プロシージャ 321, 420
　条件判断 307
　速度優先の最適化 425
　ループ 313
　論理式 303
整数
　速度の最適化 424
　メモリ使用量の節約 424
静的変数 350, 421
セットアップ プログラム 3
セパレータ バー 162
セミコロン (;)..... 147
[全ウォッチ解除] コマンド ([ﾃﾞﾊﾞｯｸﾞ] メニュー) 361, 374, 435
宣言
　Quick ライブラリ 342
　プロシージャ 335
[全ブレーク ポイント解除] コマンド ([ﾃﾞﾊﾞｯｸﾞ] メニュー).... 361, 370, 435

ソ

速度
　最適化 425
ソース コード
　印刷 133

タ

ダイアログ ボックス 107
　vs. サブメニュー 167
　あらかじめ用意されているダイアログ ボックス 109
　カスタマイズ 114
　コモン ダイアログ ツールキット 214
　作成 107
　モーダル vs. モードレス 108
タイトル バー 5
タイマー イベント 102, 104, 105
タイマー コントロール 59, 102
　初期化 103
　フォームへの配置 103
　プロパティ 103
タブ
　TAB 関数..... 148
タブ オーダー
　コントロールの削除 55
　デフォルト 54
　変更 54
ダブル クォーテーション マーク 141
ダブルクリック
　→「DblClick イベント」を参照

チ

チェック ボックス 59, 68
　作成 72
チェック マーク 165
中断 368
重複
　変数 349

ツ

[次のステートメント] コマンド ([表示] メニュー) 363, 433
[次のステートメント設定] コマンド ([ﾃﾞﾊﾞｯｸﾞ] メニュー)..... 361, 376, 435
[次を検索] コマンド ([検索] メニュー)..... 433
ツールキット
　→「コモン ダイアログ ツールキット」,「マウス ツールキット」を参照
ツール ボックス 12, 19, 37

[ツール ボックス] コマンド ([ｳｨﾝﾄﾞｳ] メニュー) 437
[ﾂｰﾙ] メニュー 12, 37, 436

テ

定数
ControlPanel プロパティの定数 298
TRUE と FALSE 306
インクルード ファイルの作成 196, 306
速度の最適化 424
定義 196
フォーム モジュール 49
プロシージャへの引き渡し 329
メモリ使用量の節約 423
ディスク キャッシュ 414
ディレクトリ リスト ボックス 59, 205
テキスト
PRINT 228
PRINT USING 230
クリップボードとのやり取り 142
書式指定 230
操作 135
ダブル クォーテーション 141
テキスト ビューポート 231
デスクトップ上の表示位置 227
ピクチャー ボックスへの表示 88
文字列の高さの指定 150
文字列の幅の指定 150
テキスト エディタ サンプル アプリケーション
シーケンシャル アクセス ファイル の例 253
シングル ドキュメント インターフェイスの例 174
テキスト ファイルの保管 251
プロパティの設定 250
マルチ ドキュメント インターフェイス (MDI) 176
メニュー コントロールの例 169
テキスト カーソル
位置の指定 238
位置の取得 239
形状の変更 239
テキスト ビューポート 231
テキスト ファイル 131
ASCII フォーム 401
コードへの読み込み 131
シーケンシャル アクセス ファイル 250
保管 128, 401
テキスト ボックス 59
[TAB] キーの使用 257
改行 136
画面 17
クリップボード 144
作成 65
入力されたデータの処理 137
複数行 136
[テキスト保管] コマンド ([ﾌｧｲﾙ] メニュー) 431
[テキスト読み出し] コマンド ([ﾌｧｲﾙ] メニュー) 431
デスクトップ 226
サンプル アプリケーション 179
数値の表示 228
テキストの表示 228
パターンの設定 300
マウス イベント 272
データ型
サイズ 419
ユーザー定義型 258
データ ファイル 241
アクセスの種類 242
オープン 244
クローズ 249
シーケンシャル アクセス ファイル 242, 249
バイナリ ファイル 242, 261
ランダム アクセス ファイル 242, 258
デバイス
操作 266
ファイル I/O vs. デバイス I/O 267
デバッグ ウィンドウ 7
[デバッグ] コマンド/ウィンドウ ([ｳｨﾝﾄﾞｳ] メニュー) 362, 437
[ﾃﾞﾊﾞｯｸﾞ] メニュー
[インスタント ウォッチ] コマンド 361, 435
[ウォッチ解除] コマンド 361, 371, 374, 435
[ウォッチ追加] コマンド 361, 373, 435
[ウォッチ ポイント] コマンド 361, 435
[エラー時中断 ON/OFF] コマンド 361, 435
[全ウォッチ解除] コマンド 361, 374, 435
[全ブレーク ポイント解除] コマンド 361, 435
[次のステートメント設定] コマンド 361, 376, 435
[トレース ON/OFF] コマンド 361, 375, 435
[ヒストリ ON/OFF] コマンド 361, 377, 435
[ブレーク ポイント ON/OFF] コマンド 361, 370, 435
デフォルトのコマンド ボタン 117

ト

等号 (=) 304
[トピック] コマンド ([ﾍﾙﾌﾟ] メニュー)..... 438
ドライブ リスト ボックス 59, 204
ドラッグ 19
ドラッグ & ドロップ..... 293
[トレース ON/OFF] コマンド ([ﾃﾞﾊﾞｯｸﾞ] メニュー)
..... 361, 375, 435
ドロップ 288

ナ

名前の付け方
　イベント プロシージャ 48
　エラー処理ルーチン 188
　コントロール 60

ニ

[～について] ダイアログ ボックスの例 180
入力
　キーボード 233
　速度優先の最適化 427

ハ

背景色
　BackColor プロパティによる変更
..... 41, 297, 405
　ControlPanel プロパティによる変更 297
バイナリ形式
　ASCII フォーム 401
　テキスト形式での保管 128
バイナリ コピーの例..... 262
バイナリ ツリーの検索..... 352
バイナリ ファイル アクセス 242
　SEEK の使い方 263
　vs. ランダム アクセス..... 261
　構文..... 261
ハイパーリンク 13
配列
　Quick ライブラリ..... 397
　静的配列と動的配列 420, 425
　プロシージャへの引き渡し 329, 335
配列識別子 417, 420
パス
　PATH 環境変数 3
　Quick ライブラリ..... 389
[パス設定] コマンド ([補助] メニュー)..... 436
パラメータ
　AS ANY キーワード 385
　vs. 引数 327
　イベント プロシージャへの引き渡し 49
　フォームとコントロールの引き渡し 332
[貼り付け] コマンド ([編集] メニュー)..... 432

ヒ

引数 327
　Button 引数 276
　DECLARE によるチェック 335
　Shift 引数 279
　vs. パラメータ 327
　値による引き渡し 333
　強制的な型変換 327
　参照よる引き渡し 332
　ビットフィールド 272
　プロシージャへの引き渡し 327
　マウス イベント..... 272
　未知の型の引き渡し 332
ピクチャー ボックス 59, 88
　テキストの出力 146
[ヒストリ ON/OFF] コマンド ([ﾃﾞﾊﾞｯｸﾞ] メニュー)
..... 361, 377, 435
日付
　書式 152
ビット単位の演算子 305
ビットフィールド引数 272
　Button 引数 276
　Shift 引数 279
表記規則
　マニュアルの表記規則 xxiv
表示
　イミディエイト ウィンドウ..... 380
[表示] メニュー
　[インクルード ファイル表示] コマンド
..... 363, 433
　[インクルード ファイル編集] コマンド
..... 363, 433
　[グリッド] コマンド 433
　[コード選択] コマンド 12, 433
　[出力画面] コマンド 363, 433
　[次のステートメント] コマンド 363, 433
　[フォーム選択] コマンド 11, 36, 433
　[メニューバー] コマンド 433
標準出力 227
表示領域 148

フ

ファイル
→「実行可能ファイル」,「モジュール」も参照
vs. モジュール 5, 126
シーケンシャル アクセス ファイル
..... 242, 249
種類 126
データ ファイル 241
バイナリ ファイル 242, 261
ファイル アクセス 242, 249
ランダム アクセス ファイル 242, 258
ファイル I/O
vs. デバイス I/O 267
バッファのサイズ vs. 文字列領域 423
ファイル アクセス
種類 242, 249
[ファイル オープン] ダイアログ ボックス
概要 216
[ファイル解放] コマンド ([ﾌｧｲﾙ] メニュー)
..... 393, 431
ファイル検索サンプル アプリケーション 208
再帰プロシージャ 352
ファイル属性
ファイル リスト ボックス 207
[ファイル追加] コマンド ([ﾌｧｲﾙ] メニュー)
..... 112, 131, 394, 431
[ファイル保管] コマンド ([ﾌｧｲﾙ] メニュー)..... 431
ファイル名
OPEN ステートメント 244
[ﾌｧｲﾙ] メニュー
[MS-DOS プロンプト] コマンド 431
[印刷] コマンド 133, 431
[終了] コマンド 12, 431
[新規フォーム] コマンド 6, 10, 18, 36, 431
[新規プロジェクト] コマンド
..... 18, 37, 122, 431
[新規保管] コマンド 401, 431
[新規モジュール] コマンド 431
[テキスト保管] コマンド 431
[テキスト読み出し] コマンド 431
[ファイル解放] コマンド 390, 393, 431
[ファイル追加] コマンド 112, 131, 431
[ファイル保管] コマンド 431
[フォーム解放] コマンド 40, 431
[フォーム追加] コマンド 431
[フォーム保管] コマンド 431
[プロジェクト オープン] コマンド
..... 37, 122, 431
[プロジェクト保管] コマンド 23, 123, 431
ファイル リスト ボックス 59, 206
フォーカス 55
フォーム 12, 17, 35
ASCII 形式での保管 401, 409
印刷 133
コントロールの移動 292
コントロールの追加 19, 37
最大化 39
削除 40
出力 146
タイマー コントロール 103
フォーカス 56
プロシージャへの引き渡し 332
プロパティの設定 323
編集 11
マウス イベント 271
メモリ消費 421
[フォーム解放] コマンド ([ﾌｧｲﾙ] メニュー)
..... 40, 431
[フォーム選択] コマンド ([表示] メニュー)
..... 11, 36, 433
[フォーム選択] ダイアログ ボックス 9, 36
[フォーム追加] コマンド ([ﾌｧｲﾙ] メニュー)..... 431
フォーム デザイナ
ASCII 形式での保管 401, 409
インストール 3
概要 6
画面 11
起動 10
機能 12
終了 12
新規フォームの作成 18
フォームの印刷 133
プログラミング環境への切り換え 6, 12
[フォーム保管] コマンド ([ﾌｧｲﾙ] メニュー)
..... 431
フォーム モジュール 17, 29, 127
$FORM メタコマンド 119, 349
PRINT メソッド 146
Quick ライブラリへの組み込み 392
イベント プロシージャ 30, 46
コード 29, 45, 49
ジェネラル プロシージャ 30, 46
実行開始ファイル 133
バイナリ形式とテキスト形式 128, 401
プロパティの共有 119, 349
モジュール レベル 29, 49, 126
復元 8
[複写] コマンド ([編集] メニュー)..... 432
プリンタ 152
ブレークポイント 370
[ブレーク ポイント ON/OFF] コマンド ([ﾃﾞﾊﾞｯｸﾞ]
メニュー)..... 361, 370, 435
フレーム 59
プログラミング環境 6
Quick ライブラリのロード 389
Quick ライブラリの作成 392, 395
アプリケーションの実行 411
インストール 3
概要 6
構成部分 5
コードの印刷 133

コードの入力 23, 46
サイズ 395
実行可能ファイルの作成 125
配列のメモリ消費 420
表示 3
フォーム デザイナへの切り換え 5, 9
プロシージャ 46
メモリ 411
メモリ マップ 415
プログラミング言語
複数言語によるプログラミング 395
プログラム
→「アプリケーション」も参照
イベントドリブン vs. 従来の Basic プログラム 32
サイズ優先の最適化 423
実行の中断 368
ステップ実行 375
速度優先の最適化 425
プログラムの実行
実行するステートメントの設定 376
実行履歴のトレース 377
ステップ 375
中断 373
ブレークポイント 373
プログラムの流れ
→「制御構造」を参照
プロジェクト 18, 121
オープン 37, 122
管理 122
起動時にオープン 123
クローズ 126
作成 18, 37, 122
フォームの解放 40
保管 123
プロジェクト ウィンドウ 4, 8
[プロジェクト オープン] コマンド
([ﾌｧｲﾙ] メニュー) 37, 122, 431
[プロジェクト] コマンド/ウィンドウ
([ｳｨﾝﾄﾞｳ] メニュー) 362, 437
プロジェクト ファイル (.MAK) 121, 126
オープン 121
解放 121
コンパイル/リンク 124
保管 128
[プロジェクト保管] コマンド ([ﾌｧｲﾙ] メニュー)
..... 23, 25, 123, 431
プロシージャ 45
DECLARE ステートメント 335
FUNCTION プロシージャ 45, 337
Quick ライブラリ 342
SUB プロシージャ 45, 332
値による引き渡し 333
イミディエイト ウィンドウからの呼び出し
..... 379, 389
オンライン ヘルプ 364
記号定数の引き渡し 329
コードの効率的な再利用 424
再帰呼び出し 351
参照による引き渡し 332
ジェネラル vs. イベント 29, 45
式の引き渡し 329
種類 45, 327
ステップ実行 375
定義の構文 325
データの引き渡し 325
配列の引き渡し 329
フォームの引き渡し 332
複数モジュール 323
変数の共有 343
変数の定義 324
変数の引き渡し 329
未知の型の引き渡し 332
ユーザー定義型の引き渡し 330
ユーザー定義型の要素の引き渡し 330
呼び出しパス 192
利点 322
プロシージャ定義
インクルード ファイル 339
構文 324
コード中の位置 336
使えないステートメント 325
パラメータ vs. 引数 327
パラメータの指定 328
プロシージャ レベルのエラー処理 188, 191
プロパティ 21
ASCII フォーム 404
Hello World アプリケーション 23
値の監視 374
イミディエイト ウィンドウでの設定 379
オブジェクトの位置 45
オブジェクトのサイズ 45
参照にかかる時間 428
実行時 21
設定 43
設定の変更 42
設定方法の比較 43
デザイン時 21, 41
デフォルトの設定 41
ファイル属性 207
複数モジュール 323
モジュール間での共有 349
プロパティ バー 12
使い方 21, 41
表示 41

へ

[ヘルプ] コマンド/ウィンドウ ([ｳｨﾝﾄﾞｳ] メニュー) 7, 362, 437
[ヘルプの使い方] コマンド ([ﾍﾙﾌﾟ] メニュー) 438
[ﾍﾙﾌﾟ] メニュー
[Visual Basic について] コマンド 438
[キーボード] コマンド 438
[索引] コマンド 438
[トピック] コマンド 438
[ヘルプの使い方] コマンド 438
[目次] コマンド 438
<編集> ボタン 7, 24, 48
[編集] メニュー
[切り抜き] コマンド 432
[消去] コマンド 432
[新規 FUNCTION プロシージャ] コマンド 432
[新規 SUB プロシージャ] コマンド 432
[貼り付け] コマンド 432
[複写] コマンド 432
[やり直し] コマンド 432
変数
STATIC 変数 350
値の保存 350
暗黙の宣言 367
イミディエイト ウィンドウでの代入 379
同じ名前 384
オンライン ヘルプ 364
共有 343
自動変数 350
重複 349
デフォルトの型 367
プロシージャへの引き渡し 329
明示的な宣言 367
モジュール間の共有 347
モジュール レベルでの共有 344
ユーザー定義型 259
ローカル 346
ローカル vs. グローバル 324

ホ

[保管条件] コマンド ([補助] メニュー) 436
[補助] メニュー
[画面設定] コマンド 436
[構文チェック ON/OFF] コマンド 364, 436
[パス設定] コマンド 436
フォーム デザイナ 435
プログラミング環境 436
[保管条件] コマンド 436
[マウス設定] コマンド 436

マ

マウス
座標の取得 293
ドラッグ 19
フォームが表示されていないとき 294
マウス イベント 271
MouseDown 273, 277, 281
MouseMove 273, 277, 281
MouseUp 273, 277, 281
ドラッグ & ドロップ 293
引数 272
[マウス設定] コマンド ([補助] メニュー) 436
マウス ツールキット 294
マウス ポインタ 274
マニュアル
ドラッグ モード 291
表記規則 xxiv
マルチ ドキュメント インターフェイス (MDI) 173

メ

メソッド 51
メタコマンド
$FORM 119, 178, 181
$INCLUDE 119, 129
メニュー
作成 156
サブメニュー 167
サンプル アプリケーション 159
セパレータ バー 162
編集 158
メニュー コントロール 59, 158
アクセス キーの割り当て 163
オン/オフの切り換え 164
コードの追加 161
実行時の追加 168
ショートカット キーの割り当て 163
セパレータ バー 162
非表示 165
プロパティの設定 158
無効化 164
メニュー デザイン ウィンドウ 12, 156
[メニュー デザイン] コマンド ([ｳｨﾝﾄﾞｳ] メニュー) 437
メニュー バー
フォーム デザイナ 12
プログラミング環境 5
メニュー項目 155
→「メニュー コントロール」も参照
[メニューバー] コマンド ([表示] メニュー) 433

メモリ
EMS 411, 412, 415
Quick ライブラリ 415
XMS 411, 412, 415
管理 411
速度との関係 428
メモリ マップ
実行可能ファイル 417
プログラミング環境 415

モ

[目次] コマンド ([ﾍﾙﾌﾟ] メニュー) 438
モジュール 126
vs. ファイル 5, 126
インクルード ファイル 339
構造化プログラミング 321
実行開始ファイル 133
テキスト形式での保管 128, 401
プロパティの共有 349
変数の共有 347
メモリ消費 421
モジュール レベル コード 29
定義の追加 49
定数の定義 49
表示 49
モジュールの保管 128
モジュール レベルのエラー処理 191
文字列
→「テキスト」を参照
文字列記述子 420
文字列定数
プロシージャへの引き渡し 329
文字列配列 420
モーダル ダイアログ ボックス 108, 180
モードレス ダイアログ ボックス 108

ヤ

[やり直し] コマンド ([編集] メニュー) 432

ユ

ユーザー定義型
DECLARE ステートメント 336
参照による引き渡し 333
プロシージャへの引き渡し 330
メモリ消費 420
要素のプロシージャへの引き渡し 330
ランダム アクセス ファイル 258
ユーティリティ 3

ヨ

呼び出しパス 192

ラ

[ライブラリ作成] コマンド ([実行] メニュー) 394, 434
ライブラリ ファイル 127, 130
→「Quick ライブラリ」も参照
オブジェクト モジュール (.LIB)
→「.LIB ファイル」を参照
ラベル コントロール 59
アクセス キー 53
作成 67
ランタイム エラー 189, 357
修正 369
ヘルプの表示 364
ランタイム モジュール
ランタイム ライブラリ 130
ランタイム ライブラリ
実行可能ファイルとのリンク 130
ランダム アクセス ファイル 242
GET と PUT の使い方 260
SEEK の使い方 266
vs. バイナリ アクセス 261
データの追加 258
レコード 258

リ

[リスタート] コマンド ([実行] メニュー) 360, 434
リスト ボックス 59, 75
vs. コンボ ボックス 75
イベント 80, 83
項目の追加と削除 78
内容へのアクセス 80
プロパティ 80, 84
メソッド 80

ル

ループ 303
DO...LOOP ステートメント 317
FOR...NEXT ステートメント 313
WHILE...WEND ステートメント 319
ループ中の不変なコード 427

レ

レコード 330
　ランダム アクセス ファイル 258

ロ

ローカル変数 324
論理エラー 358, 359
論理演算子 304
論理式 304

ワ

ワイルド カード文字 207
ワード ラップ 135

Microsoft® Visual Basic™
Programming System for MS-DOS®
プログラミング ガイド
1993 年 2 月 1 日　初版第 1 刷発行

©1993 Microsoft Corp.

発行　マイクロソフト株式会社

Printed in Japan
落丁・乱丁はお取り替えいたします。

Microsoft®

046-100-902